SAMS

22541

Commodore 128®
Assembly Language
Programming

Mark Andrews

Commodore 128®
Assembly Language
Programming

Commodore 128®
Assembly Language
Programming

Mark Andrews

Howard W. Sams & Co.
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

FIRST EDITION
FIRST PRINTING—1986

International Standard Book Number: 0-672-22541-7
Library of Congress Catalog Card Number: 86-62226

Acquisitions Editor: *Jim Hill*
Editor: *Susan Pink Bussiere*
Designer: *T. R. Emrick*
Illustrator: *Wm. D. Basham*
Cover Artist: *Dick Brooks*
Compositor: *Shepard Poorman Communications, Indianapolis*

Printed in the United States of America

Trademark Acknowledgements

All terms mentioned in this book that are known to be trademarks or service marks are listed below. In addition, terms suspected of being trademarks or service marks have been appropriately capitalized. Howard W. Sams & Co. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Commodore 64 and Commodore 128 are registered trademarks of Commodore Electronics, Limited.

Commodore 64 Macro Assembler Development System is copyrighted by Commodore Business Machines, Inc.

CP/M is a registered trademark of Digital Research, Inc.

Merlin 128 is a trademark of Roger Wagner Publishing, Inc.

Total Software Development System is a trademark of NoSync.

To my beloved teacher: Gurumayi Chidvilasananda

Contents

Preface

Why is this book different from all other books on Commodore 128 assembly language? Let us count the ways.

1. There are other books about 6502 assembly language, and there are other books about Commodore 128 machine language. But this book combines everything you need to know about 6502 assembly language, Commodore 128 machine language, and Commodore 128 assembly language. Many books that claim to teach C-128 assembly language are actually about C-128 machine language—and the difference between machine language and assembly language is very important. Assembly language is used by professional programmers to write commercial programs for the Commodore 128. Machine language, though adequate for writing short routines to incorporate into BASIC programs, is simply not the language to use if you want to write high-performance Commodore 128 programs.
2. We'll not only cover the C-128 monitor (as do most books on C-128 programming), but also learn how to use several of the assembler/editor systems that professional programmers use to write assembly language programs. The C-128 monitor is a useful programming utility, and this book covers it in detail. However, the focus is primarily on the programming tools that professional program designers use on the job.
3. This book contains a large collection of assembly language routines—many graphics related—that are useful, as well as interesting and entertaining. By typing, assembling, and saving these programs as you learn C-128 assembly language, you'll have a library of useful (and perhaps essential) assembly language routines that can be incorporated easily into your own BASIC and assembly language programs.

   You'll learn how to create your own character sets, and how to use joysticks and mouse controllers in text and high-resolution programs. You'll learn how to take advantage of the C-128's advanced capabilities by using programming techniques such as bank switching, interrupts, and self-modifying assembly language routines. You'll learn how to use the C-128's 80-column screen to display both text and high-resolution graphics, and how to write programs that use music and sound. And, as a bonus, there's a collection of interactive tutorial programs—programs for converting numbers from one base to another, intermixing BASIC and machine language programs, and more. So, when you finish your study of assembly language using *Commodore 128 Assembly Language Programming,* you'll be well on your way to becoming an expert C-128 assembly language programmer.

4. As you read this volume, you may notice that it's in English, not computerese. It was written by a Commodore owner for other Commodore owners—not by an electronics professor for engineering students and professional programmers (although many of them could learn something from it, too). If you understand BASIC—even a little BASIC—you'll be able to understand *Commodore 128 Assembly Language Programming.*
5. Finally, the format, from start to finish, is tailored to make the study of assembly language as painless as possible.

Introduction

Before you begin your quest to learn Commodore 128 assembly language, you need a few pieces of equipment. I assume that you have access to the most important item, a Commodore 128 computer. And, because most of the utility packages used in assembly language programming are disk based, you also need a Commodore compatible disk drive. A Commodore 1520 or 1525 printer, or any other line printer that's compatible with the Commodore, will also come in handy. It doesn't have to be a letter-quality printer, but it should print readable listings of assembly language programs.

You might also put a few standard reference books by your side. Some useful ones are the *Commodore 128 System Guide,* which comes with your computer, the *Commodore 128 Reference Guide for Programmers,* written by David L. Heiserman and published by Howard W. Sams & Co., Inc., and the *Commodore 128 Programmer's Reference Guide.* Other worthwhile books are listed in the Bibliography.

Finally, you'll need a special kind of software package called an assembler/editor system. An assembler/editor, often referred to simply as an assembler, is an indispensible utility for an assembly language programmer—or a student of assembly language programming. So if you don't have an assembler, you'd better buy one. You can use any C-128 or C-64 assembler, but it's best to own a Merlin 128 assembler, a TSDS assembler from NoSync, or a Commodore 64 Macro Assembler System, because these three assembler/editor packages were used to write most of the programs in this book.

If you're intrigued by the prospect of learning assembly language, then you're probably anxious to start. And I'm anxious to start helping you learn assembly language: the most advanced, exciting, and rewarding of all computer programming languages.

So what are we waiting for? Let's move on to chapter 1, and start programming in assembly language!

Part One

Principles and Techniques of Assembly Language Programming

Since you're reading this book, there's a good chance that you're interested in learning Commodore 128 assembly language. And there's no better time to start than right now. So, in listing 1-1, you'll find a program that you can type, assemble, and execute immediately using the computer's built-in machine language monitor. (If you don't know how to use the C-128 monitor, you'll have to wait, but not for long; before you're finished with this chapter, you'll be provided with a brief but handy crash course on how to use the computer's built-in monitor.)

Listing 1-1 COLORME64.OBJ program

```
00C00  A9 0E     LDA #$0E
00C02  8D 20 D0  STA $D020
00C05  A9 06     LDA #$06
00C07  8D 21 D0  STA $D021
00C0A  A9 0E     LDA #$0E
00C0C  85 F1     STA $F1
00C0E  60        RTS
```

Whether you type the COLORME64.OBJ program now or after reading the instructions that follow, I suggest that you save it under the name COLORME64.OBJ because that's the name we'll use when we refer to it later. When you've typed, assembled, and saved the program, return to BASIC and put your computer into 40-column mode (if it isn't there already). Then you can execute the COLORME64.OBJ program using the BASIC command:

```
SYS 3072
```

and you'll see immediately how it got its name.

Not for Experts Only

Now, for those who need it, here is a brief tutorial on using the Commodore 128 monitor. These instructions are not exhaustive; they are designed to provide you with just the amount of knowledge you'll need to type, assemble, save, and run the COLORME64.OBJ program. In chapter 4, we'll take a closer look at the C-128 monitor.

What's a Machine Language Monitor?

A machine language monitor is a utility that can be used to write, edit, save, and load machine language programs. A monitor can also do other kinds of jobs, but we won't concern ourselves with those now. For the moment, it's sufficient to point out that the C-128 monitor is a somewhat limited but very handy programming tool that's good to have when you want to write, run, or edit a short, simple machine language or assembly language program. It is no substitute for more specialized assembly language utilities such as assemblers or debuggers, but it can save you a lot of time and energy. In chapter 4, we'll take a closer look at the C-128 monitor, and we'll also explore the differences between assemblers and machine language monitors.

How to Activate and Deactivate the Monitor

There are two easy ways to get the C-128 monitor up and running: from a cold start or from BASIC. To invoke the monitor from a cold start, merely hold down the shift key while you turn on the Commodore. The C-128 will then "wake up" in its monitor mode. To activate the monitor from BASIC, just type the command:

```
MONITOR
```

To turn off the monitor and return to BASIC, you can type:

```
X
```

(X is the monitor's "eXit" command.)

When you turn on the C-128 monitor, you should see a screen display like the one in figure 1-1. The cryptic letters in the first line are abbreviations for six important registers, called *internal registers,* which are the main ingredients of the C-128's *central processing unit* (*CPU*). As we will see in chapter 3, a CPU is the heart—or, perhaps more accurately, the brain—of a computer. The CPU that's built into the C-128 is a large-scale integration (*LSI*) chip called the 8502. The six internal registers shown in figure 1-1 are what enable the computer's 8502 chip to do all of the wonderful things that a computer microprocessor can do. These registers, as abbreviated in figure 1-1 (and on your computer's screen), are: the program counter (PC), the processor status register (SR), the accumulator (AC), the X register (XR), the Y register (YR), and the stack pointer (SP).

Figure 1-1
C-128 monitor's startup display

```
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8
```

Only two of the 8502's registers are used in the COLORME64.OBJ program, so they are the only ones we will discuss in this chapter. But all six registers are examined in detail in chapter 3.

Program Counter and Accumulator

The two registers used in the COLORME64.OBJ program are the program counter (PC) and the accumulator (ACC). As it turns out, these are also the two busiest registers in the the 8502. The program counter rarely gets a nanosecond of rest; when a program is running, the PC has to keep at least one step ahead of the 8502 at all times because it always holds the address of the *next* instruction (or the next piece of data) to be processed in a machine language program. The accumulator is as busy as the program counter and sometimes even busier; it handles every arithmetical and logical operation that the 8502 is called upon to perform.

As we shall see in chapters 2 and 3, the accumulator in the 8502 is an 8-bit register, and the program counter is a 16-bit register. For reasons that will be explained in depth in those chapters, this means that the accumulator can hold any value ranging from 0 to 255 (or from $00 to $FF in hexadecimal notation), and the program counter can hold any value ranging from 0 to 65,535 ($0000 to $FFFF in hexadecimal notation). In 8502 assembly language, by the way, a dollar sign indicates that the number that follows is a hexadecimal number. If you're not quite sure what a hexadecimal number is, don't worry; the hexadecimal system is discussed in detail in chapter 2.

C-128 Memory Banks

Now take a close look at figure 1-1, and you will see that when the C-128 monitor is turned on, the value listed under the heading PC is $FB000. If you're an old hand at working with hexadecimal numbers, you'll probably notice that $FB000 is a five-digit (or 20-bit) hexadecimal number, and thus has one more place (or one more byte) than the 8502's 16-bit program counter can hold. So how, you may ask, can this be? How can a 20-bit number like $FB000 be stored in a program counter that is only 16 bits long?

Well, despite what the display on your screen seems to show, the contents of the program counter when you turn on the monitor are not $FB000, but $B000. The F that precedes the number B000 is not stored in the program counter; it is an *offset* that is placed on the screen to show which of 16 *memory banks* is being accessed by the 8502. To understand what this means, you need to know something about the memory configuration of the Commodore 128. So let's pause for a moment to take a brief look at the memory layout of the C-128, a topic that is discussed in more detail in chapter 10.

Memory Banks and Memory Blocks

The Commodore 128 has two 64K blocks of RAM (random-access memory) and one 48K block of ROM (read-only memory). These three memory blocks are labeled block A, block B, and block C. The screen map showing the configurations of these memory blocks is in figure F-1 (in appendix F). We will examine these memory blocks more closely in chapter 10.

To help programmers access memory blocks A, B, and C easily and conveniently, the computer is also equipped with 16 preset memory configurations called *banks.* The use of the word *bank* is a little misleading because the 16 memory banks in the C-128 are not 16 continuous blocks of memory. Instead, they are 16 different arrangements of memory segments chosen from memory block A, memory block B, and memory block C.

The C-128's 16 banks of memory, numbered from 0 to 15 in BASIC and from $0 to $F in hexadecimal (or hex), work something like a menu in a Chinese restaurant; each is made up of a mixture of whatever RAM and ROM it needs from memory block A, memory block B, and memory block C. Of the 16 memory banks built into the C-128, there are currently only four that are commonly used: banks 0 and 1, which are made up mostly of RAM from memory blocks A and B, and banks 14 and 15, which are made up primarily of ROM from memory block C. A chart showing the configurations of memory banks 0, 1, 14, and 15 (or banks 0, 1, E and F in hexadecimal notation) is in figure F-2 (in appendix F). A fuller description of this memory map is presented in chapter 10.

Bank Switching

Each of the C-128's 16 memory banks has been designed for a specific purpose; for example, a program may use one bank for calling kernel routines, another bank for running BASIC routines, and still another bank when certain ROM cartridges are plugged in. If you like, you can switch from any bank to any other bank at any time using a technique called *bank switching.*

Bank switching is used quite often in assembly language programming, and it is a particularly important part of C-128 programming because of the way the computer is designed. Although the C-128 is built around an 8-bit microprocessor, its unusual banking structure makes it capable of accessing much more memory than an 8-bit chip was designed to handle. Without bank switching, the C-128 would be just another 64K computer. With bank switching, it can access more than 128K of memory, although it can do so only 64K at a time.

For switching from bank to bank in BASIC programs, BASIC 7.0 —the version of Commodore BASIC that's built into the 128—has a special BANK command. A number of other bank-switching techniques, some of them quite complicated, are also available to the C-128 assembly language programmer. These techniques are explored in detail in later chapters, particularly in chapter 10.

Bank 15 (F) and the C-128 Monitor

For now, though, let's return to our discussion of the Commodore 128 monitor and see exactly why the number $FB000 appears on the screen when the monitor is first turned on. Both the monitor and the BASIC interpreter are situated in the computer's ROM block, memory block C. And the monitor and BASIC can both be accessed from either bank 14 or bank 15. Banks 14 and 15 are alike, except for memory addresses $D000 through $DFFF, which hold input/output routines in bank 15 and hold character generator data in bank 14. For reasons that are explained in chapter 10, access to character generator data in bank 14 is not needed when BASIC or the monitor is in use, so bank 15 is the memory bank that is most often used for accessing BASIC and the monitor. Bank 15 contains mainly BASIC ROM. However, the ROM routines that drive the monitor also appear in bank 15, beginning at memory address $B000.

When the monitor is turned on, the program that is called first is the monitor program itself. So, when the monitor is called and the address of the program being accessed appears on the screen, the address shown is $FB000—the starting address of the ROM that runs the monitor. However, when a program is *written* using the monitor, the monitor ordinarily assembles the program into another memory bank—bank 0—one of the two banks that contains mostly RAM. As we will see in chapter 10, bank 0 is usually a good place for storing user-written programs. At the discretion of the programmer, however, programs can also be stored in other banks (usually in bank 1).

Writing a Program Using the C-128 Monitor

Now we're ready to resume our crash course in how to use the Commodore 128 monitor. The C-128 monitor, like the monitors used with most microcomputers, is designed to carry out instructions written as one-letter commands. When the command A (for "assemble") is issued, for example, the monitor accepts and assembles the assembly language instructions typed at the keyboard. If your monitor is up and running, you can use the A command to assemble the COLORME64.OBJ program. To start assembling the program, type the following line:

```
A 00C00 LDA #$0E
```

Before we continue, let's decipher that line. As we have just seen, the instruction A at the beginning of the line tells the monitor to assemble a line of code. The next group of characters—00C00 (no dollar sign needed here)—instructs the monitor to start assembling source code in memory bank 0, starting at memory address 0C00. Because bank 0 is the memory bank into which programs generated by the monitor are assembled by default, it really isn't necessary to type the 0 that precedes memory address 0C00. But, because it never hurts to make everything in a program as clear as possible, bank 0 is specified in the first line of the COLORME64.OBJ program.

When the bank number and the starting address of a program have been typed in, the program itself can be typed and assembled. In the COLORME64.OBJ program, the first group of executable code is:

```
LDA #$0E
```

which means "load the accumulator with the number $0E" (or the number 14 in decimal notation). So, when the COLORME64.OBJ program is executed, the first thing that happens is that the number 14 ($0E in hex) is stored in the 8502 register called the accumulator. As mentioned, the accumulator's main job is handling arithmetical and logical functions. But the accumulator is also used quite often as a temporary storage area for data being copied from one memory address to another. And that is how the accumulator is used in the COLORME64.OBJ program.

The dollar sign preceding the number 0E, as you may recall from a few paragraphs back, shows that the number is written in hexadecimal notation. And the # symbol that precedes the dollar sign means that the number will be interpreted not as a memory address, but as a literal number. If $0E was not preceded by the # symbol, the accumulator would be loaded with the *contents of memory address* $0E. Because the # symbol does appear, however, the accumulator is loaded with the literal number $0E.

The second statement in the COLORME64.OBJ program is:

```
STA $D020
```

In 8502 assembly language, this means "store the value of the accumulator into memory address $D020." So, the STA instruction stores the value of the accumulator in memory address $D020. (The value of the accumulator does not change when this procedure takes place, so whatever is in the accumulator before the STA instruction is issued will still be there after the instruction has been carried out.)

Now let's review what happens when the first two lines of the COLORME64.OBJ program are executed. First, the statement LDA #$0E loads the value $0E into the accumulator. Then, the statement STA $D020 copies the value of the accumulator—which is now the number $0E—into memory location $D020.

How the Program Works

Before we move on to the next line of the program, let's take a moment to examine how the program works, and what it is supposed to do. As we shall see when we reach part 2, which deals with C-128 graphics and sound, the Commodore 128 can display 16 colors on its screen. In assembly language, as in BASIC, each of these colors has a distinct code number. And, as we will see in charts presented in part 2, $0E (or 14 in decimal notation) is the code number that equates to light blue. The C-128 also has several memory addresses that can be used to determine what colors are displayed on various parts of the screen. For example, memory address $D020 is used to control the color of the border around the C-128's 40-column screen; to change the color of the screen border, all a program has to do is store the desired color code in memory address $D020.

Now you know what the first two lines of the COLORME64.OBJ program do; they color the C-128's 40-column screen border light blue. And the next four lines of the program carry out similar functions. The statements LDA #$06 and STA $D021, which appear in the third and fourth lines of the program, first load the accumulator with the value #$06 (just plain 6 in decimal notation), and then store that value in memory address $D021. This operation colors the screen background dark blue. Then, in the fifth and sixth lines of the program, the statements LDA #$0E and STA $F1 change the color of the letters on the screen to light blue. So, when the COLORME64.OBJ program is assembled and executed, it does what its name implies; it colors the C-128's screen light blue and dark blue, duplicating the screen colors of the Commodore 64.

RTS Instruction

The last instruction in the COLORME64.OBJ program is RTS, which means "return from subroutine." In 6502/8502 assembly language, this instruction serves a double function. When RTS is used as the last instruction in a subroutine, it works like the BASIC instruction RETURN; it ends the subroutine and returns control of the program being processed to the instruction following the one that called the subroutine.

But when RTS is used as the last statement in an assembly language program, it returns control of the C-128 to whatever system was in operation before the program was executed—usually, the computer's BASIC interpreter. So, if BASIC is running when a machine language program is called and if the program ends with an RTS instruction, the RTS instruction ends the program and returns control of the C-128 to the computer's built-in BASIC interpreter. Because the COLORME64.OBJ program ends with an RTS instruction, this is exactly what happens when the program is executed using the BASIC command SYS 3072. After it changes the colors on your C-128's screen, the program ends and returns control to BASIC.

Assembling the Program

Now that you understand how the COLORME64.OBJ program works, you know just about all you'll need to know to type and assemble the program. But before we do that, it may be helpful to make note of something that happens each time a line of source code is typed in using the C-128 monitor. The first time you type a line of code and press Return, you may be surprised to see that your typed line disappears and is replaced with a line that looks like this:

```
00C00   A9   0E      LDA #$0E
```

Since we have discussed bank numbers and memory addresses, you can probably figure out the meaning of 00C00, the first group of typed characters in the preceding line. It means that the program being typed will be assembled starting at memory address $0C00 in bank 0. (Detailed guidance on where to store programs in memory is provided later, primarily in chapter 10. For now, it's sufficient to note that one block of memory into which short programs are often stored is the block that starts at $0C00 in bank 0.)

That takes care of 00C00, the first group of characters in the line. We also know the meaning of the last two groups of characters, LDA and #$0E, because they were explained a few paragraphs back. But how about the second and third groups, namely, A9 and 0E?

Although they are not preceded by dollar signs, A9 and 0E are—as you may guess—hexadecimal numbers. In the COLORME64.OBJ program, the values A9 and 0E are the machine language equivalents of the assembly language values LDA and #$0E. The number A9 is the equivalent of the assembly language instruction (or *operation code*) LDA. The value 0E means the same thing in machine language that it does in assembly language; it's the object, or *operand,* of the assembly language instruction, or *mnemonic,* LDA.

Machine Language Demystified

To understand how the C-128 monitor translates assembly language programs, or *source code,* into machine language programs, or *object code,* it helps to know something about what a computer does when it executes a machine language program.

As we have seen, when you enter an assembly language statement such as LDA #$0E into the C-128 monitor, the monitor displays the machine language equivalent of the statement on the screen. But that is not all that the monitor does. After the statement is converted into machine language, the monitor also stores the machine language version of the line in memory, beginning at the memory address specified when the first line is typed. So the line that appears on the screen after the statement is typed—the line:

```
00C00   A9   0E      LDA #$0E
```

—tells us not only that the assembly language instruction LDA #$0E has been typed into the monitor, but also that its machine language equivalent—the statement A9 0E—has been stored in memory in bank 0, beginning at memory address $0C00.

Now let's pause to take a look at what the 8502 chip does when it starts executing the COLORME64.OBJ program. First, the 8502 has to be given some kind of preliminary instruction—for example, a SYS command issued from BASIC—so that it knows where in the C-128's memory to look for the program that it should run. Then it goes to the memory address that it has been given and carries out whatever machine language instruction is stored in that memory location. For example, if a program is loaded into memory beginning at $0C00 when a SYS command is issued, and if the SYS command indicates that the program starts at $0C00 (or 3072 in decimal notation), then the 8502 chip will go to memory address $0C00 and carry out whatever machine language instruction it finds at that address.

Now let's assume that the COLORME64.OBJ program has been loaded into memory starting at $0C00 and that the command SYS 3072—or the command SYS DEC("0C00")—is issued from BASIC. The 8502 chip goes to memory address $0C00 and carries out whatever machine language instruction begins at that address—in this case, the instruction A9.

We have already seen what the value A9 means in Commodore 128 machine language; like the assembly language mnemonic LDA, it stores a value in the 8502 register known as the accumulator. But the machine language instruction A9, like the assembly language instruction LDA, is only half of a statement. In machine language and assembly language jargon, it is an *operation code* (*op code*), which must be followed by an *operand.*

In the COLORME64.OBJ program, the operand of op code A9 is the value 0E. Because op code A9 and operand 0E go together, they are stored in consecutive memory addresses; op code A9 is stored in memory address $0C00 and operand 0E is stored in memory address $0C01. So, when the COLORME64.OBJ program is assembled into machine language, the machine language equivalent of the statement LDA #$0E is assembled into the machine language values A9 and 0E, and those values are stored in memory addresses $0C00 and $0C01, respectively. Then, when the program is executed, the 8502 processes the two instructions together and stores the value #$0E in the accumulator.

Assembling the Rest of the Program

Now we're ready to type and assemble the rest of the COLORME64.OBJ program. And after we do that, we'll be ready to save the program, execute it, and see what it does when it runs.

Here, again, is the first line of the program:

```
00C00   A9   0E      LDA #$0E
```

After you type that line and press the Return key, your C-128 monitor will move to the next line on the screen and display the value:

```
00C02
```

This tells you the value that the 8502 program counter will hold after it executes the statement LDA #$0E; in other words, $0C02 is the memory address into which the next instruction in the program will be assembled. It's also the C-128 monitor's way of telling you that it has assembled a line of source code and is ready for you to type another line.

Now let's examine the rest of the COLORME64.OBJ program. Listing 1-2 is the assembly language version of the program.

Listing 1-2 COLORME64.OBJ program (assembly language version)

```
LDA #$0E
STA $D020
LDA #$06
STA $D021
LDA #$0E
STA $F1
RTS
```

Now that we have the entire program listed again, it might be a good time to review what it does. The first two lines of the program store the value $0E (or decimal 14) into memory address $D020, coloring the screen border light blue. The next two lines store the value 6 in memory address $D021, coloring the screen background dark blue. In the next two lines, the value $0E (decimal 14) is stored in memory address F1, coloring the characters on the screen light blue. And the last instruction, RTS, returns control of the C-128 to its BASIC interpreter.

Other Monitor Functions

When you have finished typing the COLORME64.OBJ program, you can list it on the screen by typing the monitor command:

```
D 0C00
```

Actually, the D command does more than just list a program; it "disassembles" a series of machine language instructions, or converts them from machine language into assembly language. Then it displays the assembly language version of the disassembled machine code on the screen.

Saving a Machine Language Program

The C-128 monitor can also be used to save and load machine language programs. The L command loads a program, and the S command saves one. Both commands must be used with a device number —almost always the number 8, the default device number for a Commodore disk drive. The S command must also be accompanied by the starting address of the program being saved, and the last memory address of the program plus one. So the formula for writing a save instruction while in monitor mode is:

S *''program name'',dn,addr1,addr2*

where *program name* is the name of the program being saved, *dn* is the address of the disk drive device number (usually the number 8), *addr1* is the starting address of the program being saved, and *addr2* is the address of the last byte in the program plus 1.

Thus, the COLORME64.OBJ program can be saved using the following S command:

```
S "COLORME64.OBJ",8,0C00,0C0F
```

Loading a Machine Language Program

After you store a machine language program on a disk, you can load the program into memory at any time using the monitor command L. The format for issuing a load instruction from the monitor is:

L *''program name'',dn*

where *program name* is the name of the program being loaded, and *dn* is the disk drive device number (usually 8). For example, after the COLORME64.OBJ program is stored on a disk, it can be loaded into memory using this command line:

```
L "COLORME64.OBJ",8
```

When you have practiced saving and loading the COLORME64.OBJ program a few times, you can exit the monitor using the X command and run the program by using the BASIC command:

```
SYS 3072
```

The C-128 can also carry out a number of other commands, all of which are discussed in chapter 4.

Differences between Assembly Language and Machine Language

Now that you know how assembly language programs work, you are almost ready to move to chapter 2—but not quite. First, we'll take a look at a few odd facts about the relationship between assembly language and machine language.

As we have seen in this chapter, there is a direct, instruction-by-instruction parallel between a program written in assembly language and the same program written in machine language. In fact, assembly language is not really a programming language; it's a notation system designed to make machine language easy to understand. When an assembly language program is assembled into machine language, every instruction (or *mnemonic*) in the assembly language program is converted to a machine language instruction that means the same thing. And each operand in the assembly language program is transferred verbatim into the machine language version of the program.

But there are a few important differences between the syntax used in 8502 assembly language and the syntax used in 8502 machine language. Before we go on to the next chapter, let's take a look at these differences.

We can illustrate the most significant differences between assembly language and machine language with the help of the COLORME64.OBJ program. The program is listed again in listing 1-3, this time in a tabulated format that may make its layout easier to understand.

Listing 1-3 COLORME64.OBJ program with column headings

```
Address   Object Code   Source Code

00C00     A9 0E         LDA #$0E
00C02     8D 20 D0      STA $D020
00C05     A9 06         LDA #$06
00C07     8D 21 D0      STA $D021
00C0A     A9 0E         LDA #$0E
00C0C     85 F1         STA $F1
00C0E     60            RTS
```

Low-Byte-First Rule

One difference between assembly language and machine language is the order in which 2-byte addresses are written. In the second line of the program in listing 1-3, for example, the assembly language statement STA $D020 is assembled into the machine language statement 8D 20 D0. The number $8D is the machine language equivalent of the assembly language mnemonic STA. But two bytes that make up the operand of the STA instruction are reversed when they are converted into machine language. This quirk arises from the fact that 2-

byte numbers in 6502/8502 assembly language programs are almost always stored in memory low byte first. This happens so often that you'll get used to it after a while.

Machine Language Operands

Another odd fact about 8502 assembly language is that an assembly language instruction can often be converted into several different machine language instructions, depending upon what kind of operand follows the assembly language instruction. An excellent example of this kind of irregularity appears in the COLORME64.OBJ program.

As you can see by examining the COLORME64.OBJ program in listing 1-3, the assembly language mnemonic LDA (which means "load the accumulator") is used three times. And each time it is used, it is assembled into the same machine language instruction: the instruction A9.

The mnemonic STA (which means "store the value of the accumulator") is also used three times in the program. But examine the program carefully, and you will see that STA is not translated into the same machine language instruction each time it is used. In the second and fourth lines of the program, STA is converted into the machine language instruction 8D. But in the next-to-last line, the same mnemonic is converted to the machine language instruction 85!

The reason for this seemingly odd fact lies in the size of the operand that follows each STA instruction in the assembly language version of the COLORME64.OBJ program. The first two times the STA instruction appears, it is followed by a 2-byte operand: first the address $D020 and then the address $D021. And in each of these cases, the mnemonic STA is converted into the value 8D in the machine language version of the program. The third time the STA intruction appears, however, it is followed by a 1-byte operand—the address $F1—and this time it is converted to the value 85 in the machine language version of the program.

It would appear, then, that the mnemonic STA is converted into one machine language value when it is followed by a 1-byte operand and into another machine language value when it is followed by a 2-byte operand. And this is indeed true. When a statement such as STA $D020 appears in an assembly language program, it means "store the value of the accumulator in the (2-byte) memory address $D020." But when a statement such as STA $F1 appears in a program, it means "store the value of the accumulator in the (1-byte) memory address $F1."

In a program written in assembly language, the sizes of the operands in these two statements have to be specified in the source code in which they appear; in a line of source code, it is obvious that an operand such as $D020 is two bytes long and an operand such as $F1 is only one byte long. But when statements such as $F1 and $D020 are assembled into machine language, the sizes of their operands are not obvious.

To see how difficult it can be to determine the sizes of operands in object code programs, take a close look at the last two lines of the machine language version of the COLORME64.OBJ program. When these two lines are assembled, the value $85 is stored in memory address $0C0C, the value $F1 is stored in memory address $0C0D, and the value $60 is stored in memory address $0C0E. As the program is laid out in listing 1-3, it isn't difficult to see that the machine code value 85 in the next-to-last line of the program is the equivalent of the assembly language mnemonic STA. But is the operand of this instruction $60F1 or just $F1?

Different Strokes for Different Operands

To clarify this ambiguity, the engineers who created 6502/8502 language decided that they would simply use different op codes for 1-byte and 2-byte operands. So, even though the mnemonic STA is always written the same way when it appears in an assembly language program, it is converted into the machine language instruction $85 when it is followed by a 1-byte operand, and converted into the machine language instruction $8D when it is followed by a 2-byte operand. This solves the problem. When the 8502 chip encounters op code $85, it knows that a 1-byte operand follows. When it enounters op code $8D, it knows that the next two bytes in the program constitute a 2-byte operand.

Memory Address or Literal Number?

The 8502 chip uses a similar technique for distinguishing between operands that are memory addresses and operands that are literal numbers. In the first line of the COLORME64.OBJ program, to cite one example, the operand of the assembly language mnemonic LDA is the literal number $0E. In the 6502/8502 assembly language column of the program listing, the # symbol before the number $0E shows clearly that it is a literal number. But look closely at the assembled version of that line, and you will see that there is no symbol before the number $0E. How, then, does the 8502 chip know that $0E is not a 1-byte memory address but a 1-byte literal number?

Again, it is the operand that makes the difference. In 6502/8502 machine language, the instruction $A9 means "load the accumulator with the following literal number," and the instruction $A5 means "load the accumulator with the following 1-byte address." So, if the first line of the program was LDA $0E (instead of LDA #$0E) the assembled version of the line would be A5 0E (instead of A9 0E) and the line would mean "load the accumulator with the value of memory address $0E," not "load the accumulator with the literal number $0E."

The mnemonic LDA can also be converted into other machine code values, depending on its operand. For example, when LDA is followed by a 2-byte memory address, it is assembled into the machine language instruction AD. In all, the assembly language mnemonic LDA can be used with seven kinds of operands—and that means that it can be converted into seven different machine code instructions, depending upon its operand. Most other assembly language mnemonics can also be assembled into differing machine language equivalents, depending upon what kinds of operands follow them in a program.

In 6502/8502 assembly language, the ways in which mnemonics and their operands are used together are known as *addressing modes.* The 8502 chip can be used in so many different addressing modes that chapter 6 is devoted solely to the topic of how they are used in Commodore 128 programming.

This brings us to the end of chapter 1. You have been provided with a broad overview of how assembly language and machine language are related, and you have had an opportunity to type, assemble, list, save, and load an assembly language program using the Commodore's built-in machine language monitor. You have also been introduced to a number of important programming concepts that we will explore in greater detail in later chapters. In chapter 2, for example, we will see how binary, decimal, and hexadecimal numbers are used in assembly language, and we will learn some easy methods for converting numbers from one base to another. In chapter 3, we will peek under the hood of the Commodore 128 and see what makes it run. Then we'll be ready to start writing some challenging assembly language programs.

2

By the Numbers

Exploring the binary, hexadecimal, and decimal notation systems

A one or a zero might not mean much to you, but to a computer it means quite a bit. In the world of assembly language programming, a *bit,* or *binary digit,* is the smallest piece of data that can be stored in a computer. When a bit is referred to in a computer program, it is usually expressed as 0 or 1. You may already know that, but you may not know why. So we will begin this chapter with a few important facts about bits, bytes, and binary numbers.

What's in a Bit?

A bit can express only two values, and in the binary number system, those two values are expressed as zeros and ones. When a bit is turned off, or *cleared,* its value is said to be 0. And when a bit is turned on, or *set,* its value is said to be 1.

Bits are extremely important in computer programming, and here's why: A computer is really nothing but a vast collection of tiny electronic switches, sometimes called *gates,* all connected in various ways. Each of these switches, at its most fundamental level, works much like a switch that controls a light. A gate, like a light switch, is always in one of two states; it's either on or it's off. If it's on, electricity can flow through it; if it's off, an electrical impulse cannot get through.

For the sake of convenience, the ones and zeros used to express the values of bits are usually written in groups. A group of 4 bits is called a *nibble* (or *nybble*), a group of 8 bits is called a *byte,* and a group of 16 bits is called a *word.* The numbering system used to express the states of bits in this fashion is the *binary system.*

As we saw in chapter 1, however, the binary system is not the only number system used in writing assembly language programs. Another system, called the *hexadecimal system,* is also used extensively in assembly language programming. We will be covering hexadecimal numbers later in this chapter. There is also a third number system that is often used in assembly language programming. But fortunately, it's one you're probably already familiar with—the *decimal system.*

So now you know that three kinds of numbers are commonly used in assembly language programming. To summarize, they are:

1. Decimal numbers, which are based on the value 10 and are written using the familiar arabic numerals 0 through 9.
2. Binary numbers, which are based on the value 2 and are written using the bit notations 0 and 1. In binary notation, because there are only two values to work with, 10 means 2, 11 means 3, and so on. We'll see how this notation system works later in this chapter.
3. Hexadecimal numbers, which are based on the value 16 and are written using the arabic numbers 0 through 9, plus the letters A

through F. In the hexadecimal notation system, the letters A through F are used as single-digit symbols for the values 10 through 15. As noted, hexadecimal numbers are often used in assembly language programs because they can be translated easily into binary numbers. Later in this chapter, we'll see how.

Prefixes $ and %

When a binary number appears in a 6502/8502 assembly language program, the prefix % often appears before it. This symbol is used to distinguish a binary number from a decimal or hexadecimal number. When a hexadecimal number appears in a program, the $ prefix indicates that it is a hexadecimal number. No special prefix is used before a decimal number; so, if a number without a prefix appears in a program, it is assumed to be a decimal number.

Figure 2-1 shows how prefixes are used to distinguish binary, hexadecimal, and decimal numbers from each other in 6502/8502 assembly language programs.

Figure 2-1 Number prefixes used in assembly language programs

Number with Prefix	Meaning of Prefix	Decimal Equivalent
%1101	Binary number	13
$1101	Hexadecimal number	4,353
1101	None	1,101

Binary Number System

Now let's take a closer look at the binary number system. When data is saved on a disk, it is stored magnetically in the form of on-and-off pulses that can be read off the disk and represented as binary numbers. When data on a disk is copied into a computer's memory, the magnetic pulses on the disk are converted into electronic impulses that are stored in memory. And when the electronic impulses stored in memory are written on paper, they are often written as binary digits, or bits—in other words, as zeros and ones.

The electronic impulses stored in a computer, and the various kinds of symbols used to represent those impulses in written form, lie at the heart of all computer operations. Inside a computer, the on-and-off pulses that we call bits cause the current flowing through various gates to fluctuate between low and high levels. When the electrical current flowing through a line falls below a predetermined level, the switch is considered off, and its state is represented as a zero in the binary notation system. When the level of the current rises above a certain level, the switch is considered on, and its state is represented as a one.

One of the most important features of assembly language is that it enables the programmer to control each bit in a computer's memory. Some high-level languages—for instance, the BASIC 7.0 language built into the Commodore 128—also provide the programmer with some bit-manipulation capabilities. But no high-level language is as versatile in this respect as assembly language.

That's about all that needs to be said at the moment about the concept of binary numbers. Now let's see how binary numbers are used in assembly language programming.

Penguin Math

One of my favorite methods of explaining the concept of binary numbers is with the help of something I call Penguin Math. Penguin Math is the number system that penguins would probably use if penguins could use numbers. To get an idea of how Penguin Math works, just imagine that you are a penguin living on Penguin Island. Now a penguin doesn't have five fingers on each hand. Instead, a penguin has just two flippers. So, if you are a penguin and are looking for something to count with, you don't have ten fingers to work with. You're stuck with just two flippers. Instead of being able to count up to 10 on your fingers, you're only able to count up to 2.

But suppose that you are an extremely smart penguin, a regular Einstein among your peers on Penguin Island. Then you might be able to figure out a way to count past 2 by using just two flippers. Here's one way that could be arranged.

Intermediate Penguin Math

Instead of counting on your flippers in the ordinary way—raising one flipper to represent the number 1 and raising two flippers to represent the number 2—you could decide to count using an entirely different system. For example, you decide to let a raised right flipper equate to the number 1, and to let a raised left flipper equate to the number 2. Then you can raise both flippers to equate to the number 3. If you did that, you can express the values 0 through 3, instead of just the values 1 and 2, using Penguin Numbers.

Binary on the Rocks

A new system of mathematics is never worth much, though, until you devise a notation system to write it down. So, if you want to get the maximum benefit out of your discovery of Penguin Math, you'll have to figure out some kind of notation system for expressing your new number system in writing. Fortunately, a system for notating Penguin Math is not too difficult to figure out. For example, you could decide to let a 0 represent an unraised flipper and a 1 represent a raised flipper. Then you could express the numbers 0 through 3, as shown in figure 2-2.

Figure 2-2
Penguin Math conversion chart

Flipper Configuration	Penguin Number	Decimal Equivalent
No flippers raised	00	0
Raised right flipper	01	1
Raised left flipper	10	2
Both flippers raised	11	3

The numbers in figure 2-2 are binary numbers. And they clearly show how a real smart bird can use two flippers to express four values—the values 0 through 3. Obviously, that's a considerable improvement over using your flippers to express only the numbers 1 and 2.

Discovering Feet

Now let's suppose that, while scratching your new number system in the ice on Penguin Island, you happen to notice that you are standing on two more flippers.

Voilá—bigger numbers!

Now, by using your two bottom flippers along with your two top flippers, you can count all the way up to 15, as shown in figure 2-3!

Figure 2-3
4-Bit Penguin Math

Penguin Number	Decimal Equivalent
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Be Fruitful and Multiply

Now let's see what might happen if romance bloomed on Penguin Island. Imagine that you, the greatest mathematician in penguin history, fall in love and get married. It probably wouldn't take you long to notice that since you have four flippers and your spouse has four flippers, the total number of flippers you now have to work with has suddenly doubled to eight.

If your spouse decides to cooperate with you in your mathematical experiments, the two of you can now sit on the ice and count even higher, using 8-bit Penguin Math, as illustrated in figure 2-4.

Figure 2-4
8-Bit Penguin Math

Penguin Number	Decimal Equivalent
0001 0000	16
0001 0001	17
0001 0010	18
0001 0011	19
0001 0100	20
0001 0101	21
0001 0110	22
0001 0111	23
0001 1000	24
0001 1001	25
0001 1010	26
0001 1011	27
0001 1100	28
0001 1101	29
0001 1110	30
0001 1111	31
0010 0000	32
...and so on.	

If you and your spouse kept on counting in 8-bit Penguin Math, you would eventually discover that by using all eight flippers now at your disposal, you can count from 0 to 255, for a total of 256 values.

That completes our brief foray into Penguin Math. It demonstrates that it is possible to express 256 values—from 0 through 255—using 8-bit binary numbers.

Back in the Real World

Now let's take a look at how Penguin Math applies to real-world assembly language programming. Examine the numbers in figure 2-5, and you'll see that in the binary system, every number that ends in zero is twice as large as the previous round number. In other words, the set bits in a binary number progress in powers of two. When the rightmost bit in a byte is set, the value of the byte is 1. Clear that bit and set the one to the left of it, and you have a 2. Clear the 2 bit and set the next one, and you have a 4. And so on.

Figure 2-5
Place values in a binary number

00000001 =	1	(2 to the power of 0)
00000010 =	2	(2 to the power of 1)
00000100 =	4	(2 to the power of 2)
00001000 =	8	(2 to the power of 3)
00010000 =	16	(2 to the power of 4)
00100000 =	32	(2 to the power of 5)
01000000 =	64	(2 to the power of 6)
10000000 =	128	(2 to the power of 7)

Figure 2-6 shows how this process works in greater detail.

Figure 2-6
Bit numbers and place values in a binary number

Bit Numbers	7	6	5	4	3	2	1	0
	1	0	1	1	0	1	0	1
Place Values	128	64	32	16	8	4	2	1

As figure 2-6 illustrates, every digit (or bit) in an 8-bit binary number has two identification numbers associated with it: a *bit number* and a *place value.* Bit numbers pinpoint the *locations* of bits in a byte, and place values determine the *values* of bits in a byte that are set.

The bit numbers in a byte are numbered from right to left in consecutive order, beginning with 0. So the rightmost bit in a byte is called bit 0, and the leftmost bit is bit 7. The place numbers in a byte are also ordered from right to left, but begin with the number 1. Each place value in a binary number has twice the value of the place value on its right. So bit 0, when set, has a value of 1; bit 1, when set, has a value of 2; bit 3, when set, has a value of 4; and so on.

Two Noteworthy Numbers

Now let's take a look at two binary numbers worthy of special mention:

%11111111 = 255
%11111111 11111111 = 65,535

Now why are those numbers so special? Well, the number %11111111, or 255 in decimal notation, is the largest 8-bit number. And it's particularly important in the world of Commodore 128 programming because the 8502 chip, the C-128's central processor unit (*CPU*), is an 8-bit chip. This means that the 8502 chip cannot deal with any number greater than 8 bits; the largest number it can handle is the 8-bit binary number %11111111 (255). When the 8502 is called upon to perform an arithmetical operation on a number that's more than 8 bits, or greater than 255, the number must be split into more than one byte. Then, the desired operation is carried out separately on each byte, with carries from one byte to another until the operation is completed. Then, the bytes in the resulting number must be joined before the number can be read. We'll see how this process works in greater detail later.

The number %11111111 11111111, or 65,535, is noteworthy because it is the largest 16-bit number. It is a particularly important number for Commodore 128 programmers because the 8502 is capable of accessing 64K of memory, or 65,536 bytes of memory (numbered 0 through 65,535), at a time. The reason that 64K of memory is expressed as 65,535 in decimal notation is that one kilobyte of memory, or one K, is equivalent to 1,024 bytes of memory, instead of an even 1,000. And the reason that 1K of memory is equal to 1,024 instead of 1,000 is that 1,024 is a nice round number (in the binary system) while 1,000 is not. In binary notation, 1,024 is equal to %0100 0000 0000 and 1,000 is equal to $0011 1110 1000. So when you are working with binary numbers, it's a lot easier to deal with multiples of 1,024 than with multiples of 1,000.

Incidentally, the spaces used in writing all of these binary numbers are there just so the numbers will be easier to read. Spaces are often inserted in the middle of binary numbers for this reason. Sometimes, for example, you might see the binary number 11111111 written as 1111 1111.

Hexadecimal Number System

Since computers "think" in binary numbers, the binary system is an excellent notation system for representing computer data. But, as you've clearly seen in this chapter, binary numbers have one serious shortcoming: they're extremely difficult to read. So, even though you have to understand binary numbers to be a good assembly language programmer, the binary system is not the system that is most often used in assembly language programming. The numeric system that you'll encounter most often in the assembly language programs in this book and beyond is the hexadecimal system. If the binary system would be a good system for penguins, the hexadecimal system would be ideal for a race of beings with eight fingers on each hand because, while binary numbers are based on the value 2, hexadecimal numbers are based on the value 16.

Hexadecimal numbers are often used in assembly language programming because they can help bridge the gap between the binary and decimal systems. Because binary numbers have a base of 2 and hex numbers have a base of 16, a series of four binary bits can always be translated into one hexadecimal digit. So a series of 8 bits (a byte) can always be represented by a pair of hexadecimal digits, and a series of 16 bits (a word) can always be represented by a four-digit hexadecimal number. You'll see very clearly how this works later in this chapter.

Writing Hexadecimal Numbers

In the hexadecimal system, the digits 0 through 9 have the same meanings that they have in the decimal system. But the hex system also has six other digits, which are expressed as the letters A through F. In the hexadecimal system, the number 10 is written as A, 11 is written as B, 12 is C, 13 is D, 14 is E, and 15 is F. So the numbers FC1C, 5DA4, and even ABCD are perfectly good numbers in the hexadecimal system. Table 2-1 shows what the numbers 1 through 16 look like when they're written as hexadecimal numbers.

Table 2-1
Decimal-to-Hexadecimal Conversion Chart

Decimal Number	Hexadecimal Equivalent
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10

Comparing Binary and Hexadecimal Numbers

Now let's examine the relationship between binary and hexadecimal numbers. Binary numbers, as we have seen, have a base of 2. Decimal numbers—the kind you learned in school—have a base of 10. Hexadecimal numbers have a base of 16. Table 2-2 is a chart comparing decimal, hexadecimal, and binary numbers.

Table 2-2
Decimal, Hexadecimal, and Binary Numbers Compared

Decimal Number	Hexadecimal Number	Binary Number
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

Advantages of Hexadecimal Numbers

As table 2-2 shows, the decimal number 16 is written as 10 in hex and 00010000 in binary, and is thus a round number in both the binary system and the hex system. And the hex digit F, which comes just before hex 10 (or 16 in decimal), is written 00001111 in binary. As you become more familiar with the binary and hexadecimal systems, you will begin to notice many other similarities between these two numeric systems. For example, the decimal number 255 (the largest 8-bit number) is 11111111 in binary and FF in hex. The decimal number 65,535 (the highest memory address in a 64K computer) is written 11111111 11111111 in binary and FFFF in hex.

Convenience of Hexadecimal Numbers

The point of all this is that it's much easier to convert between binary and hexadecimal numbers than it is to convert between binary and decimal numbers. And this is especially true when you're dealing with 16-bit numbers. Figure 2-7 shows how easy it is to convert between the binary and hexadecimal systems, and how much more difficult the relationship is to detect between a decimal number and its equivalents in the other two systems.

Figure 2-7
Binary, hex, and decimal numbers converted and compared

Binary:	1111	1100
Hexadecimal:	F	C
Decimal:	252	
Binary:	0010	1110
Hexadecimal:	2	E
Decimal:	46	
Binary:	1011	1000
Hexadecimal:	B	8
Decimal:	184	
Binary:	0001	1100
Hexadecimal:	1	C
Decimal:	28	

As figure 2-7 illustrates, a 4-bit binary number can always be written as one hexadecimal digit, and an 8-bit binary number can always be written as two hexadecimal digits. But there is no clear relationship between the length of a binary number and the length of the same number written in decimal notation.

This principle can be extended to longer numbers. For example, the 16-bit binary number 1111 1100 0001 1100 can be written in hex as $FC1C—a four-digit number. And, when you become familiar with the hexadecimal system, it becomes quite easy to look at a binary number like 1111 1100 0001 1100 and convert it in your head to $FC1C. But in the decimal system, the binary number 1111 1100 0001 1100 equates to 64,540—a value that has no easy-to-detect relationship with its binary equivalent. This point is illustrated in figure 2-8.

Figure 2-8 Three ways to represent a 16-bit number

Binary Number:	1111	1100	0001	1100
Hexadecimal Equivalent:	F	C	1	C
Decimal Equivalent:	64,540			

Converting from One System to Another

Because hexadecimal, decimal, and binary numbers are all used extensively in assembly language programming, it would be handy to have some kind of tool to convert numbers back and forth among these three number systems. Fortunately, a number of these tools are available. Some examples follow.

Programmer's Calculators

There are some calculators that can perform decimal-to-hexadecimal and hexadecimal-to-decimal conversions in a flash, and can even add, subtract, multiply, and divide both decimal and hexadecimal numbers. For example, Texas Instruments makes an extremely useful decimal/hexadecimal calculator called the Programmer. Several other companies also manufacture decimal/hexadecimal calculators, but the TI Programmer is the most versatile and easy-to-use model that I have found. Many assembly language program designers use the TI Programmer, or some similar calculator, and wouldn't dream of trying to get along without it. If you get the chance to buy the TI Programmer or some other good decimal/hexadecimal calculator, do it! It won't cost much, and it will be well worth the expense.

Books and Charts

There are also many books and charts you can consult when you want to convert numbers from one notation system to another. You'll find a few such charts in this chapter, and you'll also find something much better: a BASIC program that automatically performs decimal-to-hexadecimal, decimal-to-binary, and binary-to-hexadecimal conversions.

BASIC 7.0's HEX and DEC Functions

If you don't have a hexadecimal calculator handy and can't find any conversion charts, it might be helpful to know that your C-128 is equipped with a BASIC function that can convert decimal numbers to hex numbers and another function that can convert hex numbers to decimal numbers. Both functions can be used either in BASIC'S immediate mode or from within a BASIC program.

The C-128's decimal-to-hexadecimal function is written using the format:

HEX$(*d*)

where *d* (for decimal) represents the decimal number to be converted. The hex-to-decimal function is written using the command:

DEC("*h*")

where *h* represents the hexadecimal number to be converted.

For example, to convert the decimal number 53280 to hexadecimal, you could type:

```
PRINT HEX$(53280)
```

(You can also use the same statement in a program.) In response, your computer prints the hex number D020 on the screen. To convert the hex number D020 to a decimal number, you could use the statement:

```
PRINT DEC("D020")
```

The computer responds by printing the number 53280 on the screen.

Before you finish this chapter, you'll be presented with a handy type-and-run program that converts binary numbers to their decimal and hex equivalents, and converts decimal numbers to hex numbers and vice versa. First, though, to get a better idea of how the binary, decimal, and hexadecimal systems work, it might be a good idea to perform some conversions the old-fashioned way—by hand.

Binary-to-Decimal Conversion

Binary-to-decimal conversion might seem difficult at first, but it isn't too tough after you get the hang of it. In a binary number, as we saw earlier in this chapter, the rightmost bit always represents the decimal number 1—which, in higher mathematical terms, can be expressed as 2 to the power of 0. The next bit to the left represents 2 to the power of 1, the next represents 2 to the power of 2, and so on. The bit positions in an 8-bit binary number are numbered 0 to 7, starting from the rightmost digit. The rightmost bit—bit 0—represents 2 to the 0th power, or the number 1. And the leftmost bit—bit 7—is equal to 2 to the 7th power, or 128. Figure 2-9 is a list of simple equations that illustrate the meaning of each bit in an 8-bit binary number.

The format used in figure 2-9 illustrates one easy way to convert a binary number into a decimal number. Here's the method: Instead of writing the binary number in its usual form, from left to right, write it in a vertical column, with bit 0 at the top of the column and bit 7 at the bottom. Next, multiply each bit in the binary number by

Figure 2-9 Values of bits in an 8-bit binary number

Bit 0	= 2 to the 0th power =	1
Bit 1	= 2 to the 1st power =	2
Bit 2	= 2 to the 2d power =	4
Bit 3	= 2 to the 3d power =	8
Bit 4	= 2 to the 4th power =	16
Bit 5	= 2 to the 5th power =	32
Bit 6	= 2 to the 6th power =	64
Bit 7	= 2 to the 7th power =	128

the decimal number that it represents. Then add the results of these multiplications. The total you get is the decimal value of the binary number.

As an example, suppose you want to convert the binary number 00101001 into a decimal number. This is how to do it:

1 ×	1	=	1
0 ×	2	=	0
0 ×	4	=	0
1 ×	8	=	8
0 ×	16	=	0
1 ×	32	=	32
0 ×	64	=	0
0 ×	128	=	0
	Total	=	41

According to the result of these calculations, the binary number 00101001 is equivalent to the decimal number 41. And, if you look up either 00101001 or 41 on any binary-to-decimal or decimal-to-binary conversion chart, you'll see that the calculation is accurate. This conversion technique works with any binary number.

Decimal-to-Binary Conversion

Now we'll reverse direction and convert a decimal number to a binary number. Here's how to do it: First, pick a decimal number—any decimal number. Then divide it by 2. Next, write down both the quotient and the remainder. Because we're dealing with a division by 2, the remainder is either 1 or 0. So what we end up writing is a quotient, followed by a remainder of either 1 or 0. Next, take the quotient, divide it by 2, and write it down. If there's a remainder (a 1 or a 0), jot that down too, underneath the first remainder.

When there are no more numbers left to divide, write down all of the remainders, reading from the bottom to the top of the list. What we have is a binary number—a number made up of ones and zeros. That number is the binary equivalent of our original decimal number.

This conversion technique is illustrated for the decimal number 117 as follows:

117 / 2 = 58 with a remainder of 1
58 / 2 = 29 with a remainder of 0
29 / 2 = 14 with a remainder of 1
14 / 2 = 7 with a remainder of 0
7 / 2 = 3 with a remainder of 1
3 / 2 = 1 with a remainder of 1
1 / 2 = 0 with a remainder of 1
0 / 2 = 0 with a remainder of 0

To perform the decimal-to-binary conversion, simply copy the binary digits in the right-hand column, writing them horizontally from right to left, with the top digit on the right. You'll then see that the binary equivalent of the decimal (not hexadecimal) number 117 is 01110101. If you have a decimal-to-binary conversion chart handy, you can use it to confirm the accuracy of this calculation.

Decimal-to-Hexadecimal Conversion

Decimal-to-hexadecimal conversion also looks rather difficult at first glance, but it really isn't too difficult to master. First, divide the decimal integer by 16. Then write down the remainder, like this:

64540 / 16 = 4033 with a remainder of 12

Next, divide the integer part of the preceding quotient by 16, and write down the result of that calculation:

4033 / 16 = 252 with a remainder of 1

Now repeat this process until you have a quotient of 0. Here's the entire set of calculations needed to convert the decimal number 64540 into a hexadecimal number:

64540 / 16 = 4033 with a remainder of 12
4033 / 16 = 252 with a remainder of 1
252 / 16 = 15 with a remainder of 12
15 / 16 = 0 with a remainder of 15

When you've finished this series of calculations, you must convert any remainder that's greater than 9 into its hexadecimal equivalent. In the preceding problem, three remainders are greater than 9: the value 12 in the first line, the value 12 in the third line, and the value 15 in the fourth line. The decimal number 12 equates to the letter C in hexadecimal notation, and the decimal number 15 equates to the letter F. So the remainders in the preceding problem, converted into hex, are:

C
1
C
F

Read these four numbers, starting from the bottom and reading up, and you have the hexadecimal number FC1C, which is the number we're looking for: the hex equivalent of decimal 64540!

This conversion process works with any decimal integer. But there is an easier way to convert a decimal number to a hexadecimal number: Let your computer do it for you, with a BASIC program! (The program is listed later in this chapter.)

Hex-to-Binary and Binary-to-Hex Conversions

To convert a hexadecimal number to a binary number, all you have to do is use a chart like the one in table 2-3.

Table 2-3 Hexadecimal-to-Binary Conversion Chart

Hexadecimal Number	Binary Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Table 2-3 shows how easy it is to convert a hexadecimal number to its binary notation: Merely convert each hex digit separately and then string them together. For example, the binary equivalent of the hexadecimal number C0 is 1100 0000. The binary equivalent of the hex number 8F2 is 1000 1111 0010. And so on.

To convert binary numbers to hexadecimal numbers, use the chart in reverse. The binary number 1101 0110 1110 0101, for example, is equivalent to the hexadecimal number D6E5.

An Easier Way

Even though it isn't difficult to convert binary numbers to hexadecimal numbers and vice versa, it is time consuming to do it by hand. And when you program in assembly language, you have to do a lot of binary-to-decimal and decimal-to-binary converting. And that's just the kind of job that a computer was cut out for. So take a look at listing 2-1, and you'll find a BASIC 7.0 program that converts binary, decimal, and hexadecimal numbers from one base to another. It's menu driven, it was written especially for the Commodore 128, and it's yours for the typing. So type, run, experiment, and enjoy!

Listing 2-1 CONVERT.BAS program

```
10 REM ***** CONVERT.BAS*********
20 DIM HEX$(8),BIT$(8),BIT(8),H$(16),B$(16),
   TEMP$(2)
30 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
40 DATA 0000,0001,0010,0011,0100,0101,0110
50 DATA 0111,1000,1001,1010,1011,1100,1101,
   1110,1111
60 FOR L=1 TO 16:READ H$(L):NEXT L
70 FOR L=1 TO 16:READ B$(L):NEXT L
80 PRINT CHR$(147):PRINT:PRINT "   NUMBER-
   BASE CONVERSION PROGRAM":PRINT
90 PRINT:PRINT "   (A) DECIMAL TO
   HEXADECIMAL"
100 PRINT "   (B) HEXADECIMAL TO DECIMAL"
110 PRINT:PRINT "   (C) BINARY TO DECIMAL"
120 PRINT "   (D) DECIMAL TO BINARY"
130 PRINT:PRINT "   (E) HEXADECIMAL TO
    BINARY"
140 PRINT "   (F) BINARY TO HEXADECIMAL":
    PRINT
150 PRINT "   (Q) QUIT":PRINT:PRINT
160 A$="":INPUT "   TYPE CHOICE (A-F OR Q)
    ";A$
170 IF A$="" THEN 160
180 IF LEN(A$)<>1 THEN 160
190 IF A$="Q" THEN END
200 IF A$<"A" OR A$>"F" THEN 160
210 A=ASC(A$)-64:REM TRANSLATE A$ INTO AN
    INTEGER FROM 1 (A) TO 6 (F)
220 IF A<1 OR A>6 THEN 80:REM IF A$<A OR
    A$>F THEN MENU
230 ON A GOTO 250,350,430,640,850,1030
240 REM ****** DECIMAL TO HEX *******
250 PRINT CHR$(147):PRINT "        DECIMAL-
    TO-HEX CONVERSION":PRINT
260 PRINT "RANGE: 0 TO 65535"
270 A$="":PRINT:PRINT "TYPE DECIMAL NUMBER
    (OR M FOR MENU)":INPUT A$
280 IF A$="" THEN 270
290 IF A$="M" THEN 80
300 IF A$<"0" OR A$>"65535" THEN 270
```

Listing 2-1 cont.

```
310 D=VAL(A$)
320 PRINT "HEX: ";HEX$(D)
330 GOTO 270
340 REM ****** HEX TO DECIMAL *********
350 PRINT CHR$(147):PRINT "      HEX-TO-
    DECIMAL CONVERSION "
360 PRINT:PRINT "         RANGE: 0 TO
    FFFF":PRINT
370 A$="":PRINT:PRINT "TYPE HEX NUMBER (OR
    M FOR MENU):":INPUT A$
380 IF A$="M" THEN 80
390 IF A$<"0" OR A$>"FFFF" THEN 370
400 IF LEN(A$)>4 THEN 370
410 PRINT "DEC: ";DEC(A$)
420 GOTO 370
430 REM ***** BINARY TO DECIMAL *******
440 PRINT CHR$(147):REM CLEAR SCREEN
450 PRINT:PRINT "      BINARY-TO-DECIMAL
    CONVERSION"
460 A$="":PRINT:PRINT "ENTER AN 8-BIT
    BINARY NUMBER"
470 INPUT "(OR M FOR MENU): ";A$:PRINT
480 IF A$="M" THEN 80
490 IF LEN(A$)<>8 THEN 460
500 FOR L=8 to 1 STEP -1
510 BIT$(L)=MID$(A$,L,1)
520 IF BIT$(L)<>"0" AND BIT$(L)<>"1" THEN
    460
530 NEXT L
540 FOR L=1 TO 8
550 BIT(L)=VAL(BIT$(L))
560 NEXT L
570 ANS=0
580 M=256
590 FOR L=1 TO 8
600 M=M/2:ANS=ANS+BIT(L)*M
610 NEXT L
620 PRINT "DECIMAL:";ANS
630 GOTO 460
640 REM **** DECIMAL TO BINARY *******
650 PRINT CHR$(147):REM CLEAR SCREEN
660 PRINT:PRINT "     DECIMAL-TO-BINARY
    CONVERSION"
670 PRINT:PRINT "         RANGE: 0 TO
    255":PRINT
680 PRINT:BM$="":A$="":INPUT "TYPE A NUMBER
    (OR M FOR MENU)";A$
690 IF A$="M" THEN 80
700 IF A$="0" THEN 830
```

Listing 2-1 cont.

```
710 IF VAL(A$)<1 OR VAL(A$)>255 THEN 680
720 NR=VAL(A$)
730 FOR L=8 TO 1 STEP -1
740 Q=NR/2
750 R=Q-INT(Q)
760 IF R=0 THEN BIT$(L)="0":GOTO 780
770 BIT$(L)="1"
780 NR=INT(Q)
790 NEXT L
800 PRINT "BINARY: ";
810 FOR L=1 TO 8:PRINT BIT$(L);:NEXT L:
    PRINT
820 GOTO 680
830 FOR L=1 TO 8:BIT$(L)="0":NEXT L:GOTO
    800
840 REM ****** HEX TO BINARY ******
850 PRINT CHR$(147):PRINT "      HEX-TO-
    BINARY CONVERSION"
860 PRINT:PRINT "            RANGE: 0 TO
    FF":PRINT
870 PRINT:PRINT "TYPE HEX NUMBER (OR M FOR
    MENU):":A$="":INPUT A$
880 IF A$="M" THEN 80
890 IF LEN(A$)>2 OR LEN(A$)<1 THEN 870
900 HEX$(1)="":HEX$(2)=""
910 FOR L=1 TO LEN(A$)
920 HEX$(L)=MID$(A$,L,1)
930 IF HEX$(L)<"0" OR HEX$(L)>"F" THEN 870
940 IF HEX$(L)>"9" AND HEX$(L)<"A" THEN 870
950 NEXT L
960 IF HEX$(2)="" THEN HEX$(2)=HEX$(1):HEX$
    (1)="0"
970 FOR L=1 TO 16:IF HEX$(1)=H$(L) THEN
    BIT$(1)=B$(L)
980 NEXT L
990 FOR L=1 TO 16:IF HEX$(2)H$=(L) THEN
    BIT$(2)=B$(L)
1000 NEXT L
1010 PRINT:PRINT "BIN: ";
1020 PRINT BIT$(1);BIT$(2):GOTO 870
1030 REM ***** BINARY TO HEX *******
1040 PRINT CHR$(147):REM CLEAR SCREEN
1050 PRINT:PRINT "       BINARY-TO-HEX
     CONVERSION"
1060 PRINT:PRINT "ENTER AN 8-BIT BINARY
     NUMBER"
1070 A$="":INPUT "(OR M FOR MENU): ";
     A$:PRINT
1080 IF A$="M" THEN 80
```

Listing 2-1 cont.

```
1090 IF LEN(A$)<>8 THEN 1060
1100 FOR L=8 TO 1 STEP -1
1110 BIT$(L)=MID$(A$,L,1)
1120 IF BIT$(L)<>"0" AND BIT$(L)<>"1" THEN
     460
1130 NEXT L
1140 BIT$=BIT$(1)+BIT$(2)+BIT$(3)+BIT$(4)
     +BIT$(5)+BIT$(6)+BIT$(7)+BIT$(8)
1150 T1$=LEFT$(BIT$,4):T2$=RIGHT$(BIT$,4)
1160 FOR L=1 TO 16:IF T1$=B$(L) THEN HEX$
     (1)=H$(L)
1170 NEXT L
1180 FOR L=1 TO 16:IF T2$=B$(L) THEN HEX$
     (2)=H$(L)
1190 NEXT L
1200 PRINT "HEX: ";HEX$(1);HEX$(2)
1210 GOTO 1060
```

16-Bit Numbers in PEEK and POKE Commands

Before we move on to chapter 3, there's one more important topic to cover: how to use 16-bit numbers in PEEK and POKE commands. As mentioned earlier, the Commodore 128 belongs to a class of computers called 8-bit computers. This means, among other things, that the C-128's CPU and memory registers cannot handle binary numbers larger than eight bits. And the largest 8-bit number, as we have seen in this chapter, is the decimal number 255.

Because of this limitation, the only way to store a 16-bit number in an 8-bit computer is to divide it into two 8-bit numbers and then store those two numbers in two consecutive memory registers.

Low-Byte-First Rule Revisited

As you may remember from chapter 1, when you store a 16-bit number in a 6502-based or 8502-based computer, the usual convention is to store the lower (or low-order) byte first and the higher (or high-order) byte second. For example, if the hexadecimal number $FC1C is stored in hexadecimal memory addresses $8000 and $8001, the nibble $FC is stored in memory register $8000, and the nibble $1C is stored in memory register $8001.

Storing a 16-Bit Number in RAM

Now let's suppose you want to store a 16-bit number in memory registers $8000 and $8001 using a BASIC POKE command. Because BASIC programs are written using decimal numbers, the first thing you'll have to do is convert the addresses you want to use—in this case, $8000 and $8001—into decimal numbers. Make that conversion (with the hexadecimal-to-decimal conversion program in this chapter, if you like) and you'll find that the decimal equivalents of $8000 and $8001 are 32768 and 32769, respectively.

After you've calculated the decimal equivalents of the addresses you want to POKE values into, you can use a BASIC program to do the actual poking. One such program, called POKE16.BAS, is shown in listing 2-2.

Listing 2-2 POKE16.BAS program

```
10 REM ****** POKE16.BAS ******
20 AL=32768:AH=32769
30 PRINT "TYPE A POSITIVE INTEGER"
40 PRINT "RANGING FROM 0 TO 65535"
50 INPUT X
60 HI=INT(X/256):LO=X-HI*256
70 BANK 0
80 POKE AL,LO:POKE AH,HI
90 BANK 15
```

Retrieving a 16-Bit Number from RAM

Now suppose you want to retrieve a 16-bit number from RAM using a PEEK command. You can do this with a BASIC program like the one presented in listing 2-3.

Listing 2-3 PEEK16.BAS program

```
10 REM ****** PEEK16.BAS ******
20 AL=32768:AH=32769
30 BANK 0
40 X=PEEK(AH)*256+PEEK(AL)
50 BANK 15
60 PRINT X
```

And that's all there is to retrieving a 16-bit number from RAM. So now, if you'd like to take a peek inside your Commodore 128, please move on with me to chapter 3.

3

In the Chips

A peek inside the 8502 microprocessor

Only two kinds of microprocessors are commonly used in 8-bit computers, and both kinds are built into the Commodore 128. The C-128's main microprocessor, or *central processor unit (CPU)*, is an 8502 chip—a new and updated version of the popular 6502 chip designed by MOS Technology, Inc. The computer also contains a Z-80A chip, which is compatible with the CP/M operating system and makes the Commodore 128 compatible with thousands of business programs.

Improved Twice Over

The C-128's main chip, the 8502, is almost identical to its predecessor, the 6510. And the 6510, which was designed for the Commodore 64, is almost identical to the original 6502 chip created by MOS Technology. The main difference between the 6502 and the 6510 is that the 6510 has some added input/output capabilities. There are only two differences between the 6510 and the 8502. One is that the 8502 is designed to handle the increased memory capacity of the C-128. The other difference is that the 8502 can run twice as fast as the 6510: at a rate of 2 MHz, compared with a rate of 1 MHz for its predecessor. But the C-128 cannot generate a screen display when it is running at double speed because the chips that produce video displays are not clocked to run at 2 MHz.

One important feature of the 8502 and its predecessor, the 6510, is that both chips are designed to be programmed with standard 6502 assembly language. This feature is noteworthy because some updated chips in the 6502 family—for example, the 65C02 chip used in the Apple IIc and the Apple IIe—are equipped with additional assembly language instructions that are not included in the standard 6502 assembly language instruction set. Because the 8502 is not equipped with any additional instructions, it can be programmed using standard 6502 assembly language. So 6502 assembly language is what we'll be discussing in this chapter and in the rest of this book.

The C-128's other main chip, the Z-80A, is a 4 MHz version of Zilog's standard Z-80 processor. A Z-80A chip was included in the C-128 as an alternate CPU primarily to enable the computer to run CP/M software. Since most CP/M programs are business related, and since very little new CP/M software is being developed these days, it is unlikely that many C-128 owners will be very interested in learning to program in Z-80 assembly language. So this book does not devote a tremendous amount of attention to CP/M assembly language programming.

For readers who are interested in learning Z-80 assembly language, a number of books dealing with the subject are available, and some are listed in the Bibliography. Also, a Z-80 programming kit containing two assemblers and put together especially for C-128 users, is available from Commodore.

All in the (6502) Family

As noted, the C-128's main CPU, the 8502, is a member of the venerable 6502 family of 8-bit microprocessors. The 6502 and its descendants have been used not only in the Commodore 128 and the Commodore 64, but also in every 8-bit Apple and Atari computer ever manufactured. According to computer lore, one of the reasons that Stephen Wozniak and Steven Jobs built the first Apple around a 6502 instead of a Z-80 was that the 6502 could be purchased for less than ten dollars. Another interesting fact is that MOS Technology, which designed the 6502, is now owned by Commodore.

What Every Computer Is Made Of

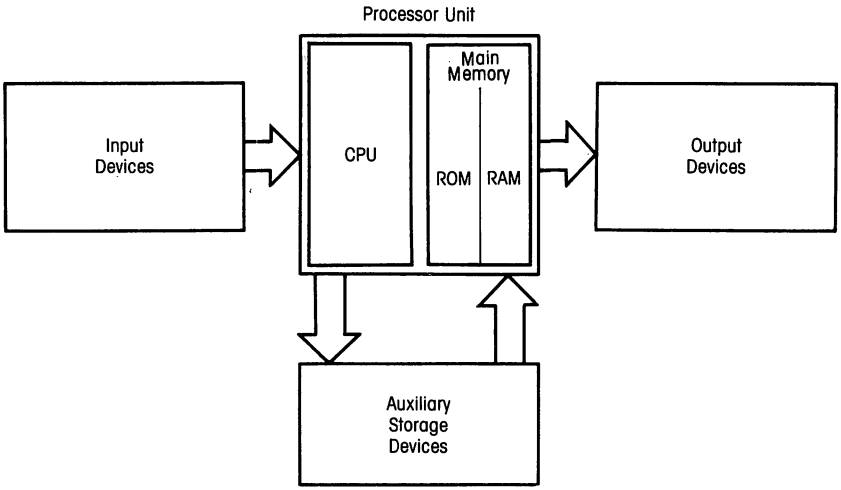
Before we finish this chapter, we'll be taking a look inside your C-128's CPU to see what makes it tick. First, though, it would probably be helpful to examine the overall architecture of a computer system. Then, when we start discussing the internal architecture of the 8502, you'll know how all the parts fit together.

According to standard texts in computer science, every computer system is made up of four kinds of components. These components are:

1. One or more *input devices.* An input device is a component designed for entering data into a computer. The most common input device is a computer keyboard. Other kinds of input devices include game controllers, mice, graphics tablets, telephone modems, and light pens.
2. One or more *output devices.* An output device, as its name implies, is a component used for getting output from a computer. The most common output devices are video monitors and printers. Other output devices include plotters, telephone modems, and loudspeakers.
3. One or more *auxiliary storage devices.* An auxiliary storage device is a component used to store data when the data is not being used by a computer. The most common data storage devices are tape and disk drives.
4. A *processor unit,* in which all computer processing takes place. The Commodore 128's processor unit is built into the keyboard. But many computers have separate processor units. Processor units come in various sizes; some can be placed on a desk, some are designed to be placed on the floor alongside a desk, and some very large processor units fill an entire room.

Figure 3-1 is a block diagram that shows the four major components of a computer system. It also shows that a computer's *processor unit* is, in turn, made up of two parts: a *CPU,* or *central processing unit,* and *main memory.* Main memory is subdivided into two parts: *random-access memory (RAM)* and *read-only memory (ROM).* Let's pause for a moment to examine these two types of memory. Then we'll move on to the CPU, the main topic of this chapter.

Figure 3-1
Architecture of a computer system



RAM and ROM

The big difference between RAM and ROM is that RAM can be erased and ROM can't. Every time you turn your computer off, everything stored in RAM promptly gets wiped out. But when you turn your computer on again, everything that was in ROM is still there.

ROM can't be erased because it is permanently etched into a bank of memory chips inside the Commodore. So it's as permanent a part of the computer as the keyboard. In computer jargon, ROM is *nonvolatile* and RAM is *volatile.*

Two very important parts of the computer's ROM are its BASIC 7.0 interpreter, which most C-128 owners would be lost without, and the block of memory that holds the computer's *operating system,* or *OS.* The Commodore's operating system is really just a long machine language program—but what a program! Part of the C-128 operating system is its *kernel*—an extensive and very useful collection of subroutines that are available for use in any machine language program. The C-128 operating system also contains built-in routines that can generate text characters, display colors, accept keyboard inputs, and operate I/O devices such as printers, video screens, and disk drives. In later chapters, we'll use many of the machine language routines that are resident in the computer's operating system, particularly those that are accessible from the C-128 kernel.

ROM, as you can imagine, was not built in a day. Your Commodore's ROM package is the result of a lot of work by a lot of assembly language programmers. RAM, on the other hand, can be written to by anybody—even you.

Most of the computer's available memory space is occupied by RAM. And the lion's share of RAM in the Commodore 128 is available for use in user-written programs. ROM is sometimes compared with an empty notebook. When you turn the computer on, the portion of memory that's dedicated to ROM is as empty as the sheets of paper in a notebook just purchased at a store. Each time you write a program, type a document with a word processor, or load a program from an auxiliary storage device, the material that you've generated is always stored somewhere in RAM.

Unfortunately, though, there's never any guarantee as to how long something that's written into RAM will remain. Turn your computer off and everything you've stored in RAM suddenly disappears. That's why Commodore and other hardware manufacturers sell so many cassette data recorders and disk drives. After you've written a program, you have to store it on some kind of mass storage medium, such as a disk or a cassette, if you don't want it erased when the power is turned off.

Your computer's RAM, or main memory, can be visualized as a huge grid made up of thousands of compartments, or cells—something like tiers upon tiers of post office boxes along a wall. Each cell in this vast memory matrix is called a memory location, or a memory register, and each memory register, like each box in a post office, has an individual and unique memory address.

The analogy between computers and post office boxes doesn't end there. A computer program, like a skilled postal worker, can get to any location in its memory about as quickly as it can get to any other. In other words, it can access any location in its memory at *random.* And that's why user-addressable memory in a computer is known as random-access memory (RAM).

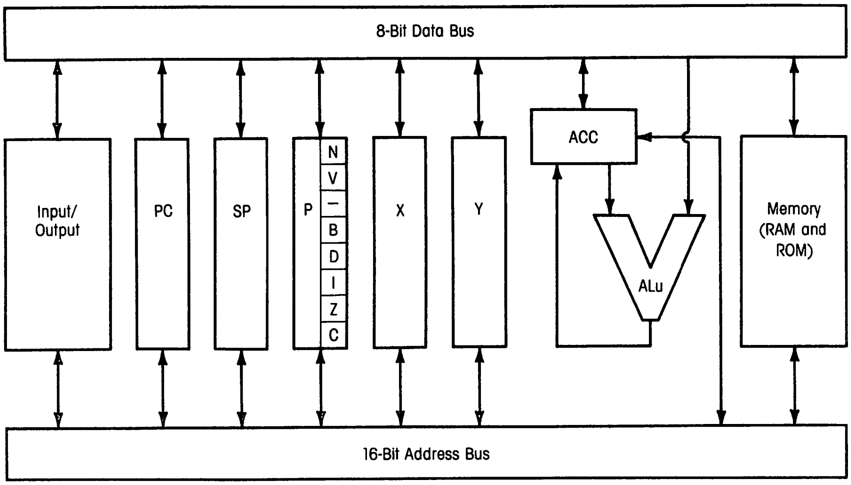
With that introduction, we are now ready to get under the hood of your Commodore 128 and take a look at how it works. Then you'll be able to find your way around inside your computer, and you'll be ready, at last, to start doing some assembly language programming.

Inside a CPU

As you may recall from chapter 1—and as figure 3-2 illustrates—the 8502 chip (the CPU) does its work with the help of six *internal registers,* or memory registers, that can be used to store and manipulate data. In addition, the 8502 contains a very important component called an *arithmetic and logic unit,* or *ALU.* The ALU, as its name implies, can perform arithmetical and logical operations. We'll see how the ALU works a little later in this chapter.

Your computer also contains a set of transmission lines called *buses.* Buses are also appropriately named; their job is to move data

Figure 3-2
Inside the 8502



back and forth between the registers in your computer's 6510/8502 chip and the memory registers in your Commodore.

There are two kinds of buses in the Commodore 128: an 8-bit data bus and a 16-bit address bus. The data bus, as its name implies, is used mainly for passing data back and forth between your computer's 8502 chip and your computer's memory registers. The address bus, as you might also guess from its name, is used to keep track of the addresses of the various memory registers used in a program.

Registers in the 8502

In addition to its accumulator, the 8502 processor has five other internal registers. They are the *X register,* the *Y register,* the *program counter,* the *stack pointer,* and the *processor status register.* Here are brief descriptions of the functions of each of these five registers:

- The X register (abbreviated X) is an 8-bit register that is often used for temporary storage of data during a program. But the X register has a special feature, too; it can be incremented and decremented with a pair of assembly language instructions (INX and DEX), and it is therefore often used as an index register, or counter, during loops in programs.
- The Y register (abbreviated Y) is also an 8-bit register, and can also be incremented and decremented with a pair of instructions (INY and DEY). So the Y register, like the X register, is used both for data storage and as a counter.
- The program counter (abbreviated PC) is actually a pair of 8-bit registers that are used together as one 16-bit register. The

two 8-bit registers that make up the program counter are occasionally referred to as the program counter low (PCL) register and the program counter high (PCH) register. The program counter always contains the 16-bit memory address of the *next* instruction to be executed by the 8502 processor. When that instruction is carried out, the address of the next instruction is loaded into the program counter.

- The stack pointer (which can be abbreviated either S or SP) is an 8-bit register that always contains the address of the next available memory address in a block of RAM called the *stack.* The 8502 stack, usually referred to simply as the stack, is a special block of memory in which data is often stored temporarily during the execution of a program. When subroutines are used in assembly language programs, the 8502 chip uses the stack as a temporary storage location for return addresses. You can use the stack for other purposes in assembly language programs, too. In chapter 6, the operation of the stack is discussed in more detail.
- The processor status register (often called simply the status register, but abbreviated P) is an 8-bit register that keeps track of the results of operations performed by the 8502. The processor status register is such an important part of the 8502 chip that we'll take a closer look at it later in this chapter.

ALU

One of the busiest components in the 8502 chip is the arithmetic and logic unit, or ALU. Every time your computer performs a calculation or a logical operation, the ALU is where the work is accomplished.

The ALU can actually perform only two kinds of calculations: addition operations and subtraction operations. Division and multiplication problems can also be solved by the ALU, but only in the form of sequences of addition and subtraction operations. The ALU can also compare values. But as far as the 8502 chip is concerned, the comparison of two numbers is also an arithmetical operation. When the 8502 chip compares two values, it subtracts one value from the other. Then, by checking the results of this subtraction operation, it can determine whether the subtracted value is more than, less than, or the same as the value from which it was subtracted.

The 8502 chip's ALU has two inputs and one output. When two numbers are to be added, subtracted, or compared, one number is put in the ALU through one of its inputs, and the other number is put in through the other input. The ALU then carries out the requested calculation, and puts the answer on a data bus so that it can be transported to another register.

ALU Hopper

As figure 3-2 illustrates, the ALU is often depicted in diagrams as a V-shaped hopper. The ALU has two inputs, which are traditionally illustrated as the two arms of the the hopper, and one output, traditionally represented as the bottom of the V.

How the ALU Works

When two numbers are added, subtracted, or compared, here's what happens. First, a number is stored in the 8502's accumulator. Next, the accumulator deposits that number in the ALU through one of the ALU's inputs. The other number is then placed in the ALU through its other input. Then the ALU carries out the requested calculation, and the result of the calculation appears at the ALU's output. As soon as the answer appears, it is placed in the accumulator, where it replaces the value that was originally stored there.

Listing 3-1, a short assembly language program titled ADDNRS.S, shows how this process works. We'll see exactly what the program does later in this chapter.

Listing 3-1
ADDNRS.S
program, version 1

```
LDA #2
ADC #2
STA $FA
```

The first statement in the ADDNRS.S program, LDA #2, means "load the accumulator with the literal number 2." As you may recall from chapter 1, the # symbol in front of the numeral 2 means that the 2 in the instruction will be interpreted as a literal number. If there was no # symbol, the 2 would be interpreted as the address of a memory register.

The second instruction in the listing, ADC, means "add the literal number 2 with carry." In 6502/8502 arithmetic, the addition of two numbers often results in a carry from a lower byte to a higher byte (in much the same way that numbers are carried from one column to another when you perform ordinary addition. If there was a carry in the ADDNRS.S program, the instruction ADC could handle it, and later in this chapter you'll find out how. But in this addition problem, there is no number to be carried, so all the ADC instruction does is add 2 and 2.

When the program reaches the statement ADC #2, the 2 that has been loaded into the accumulator is deposited into one of the ALU's inputs. The instruction ADC #2 is placed in the ALU's other input. The ALU then carries out this instruction; it adds 2 and 2, and places the sum back in the accumulator.

Now we're ready for the third and last instruction in this program. The numbers 2 and 2 have been added, and their sum is now in the accumulator. The instruction in line 3, STA, means "store the contents of the accumulator" (in the memory address that follows).

Because the accumulator now holds the value 4 (the sum of 2 and 2), the number 4 will be stored somewhere.

As you can see, the memory address that follows the STA instruction is $FA—the hexadecimal equivalent of the decimal number 250. So it appears that the number 4 will be stored in memory register $FA. Now take a close look at the hexadecimal number $FA in line 3. Because there is no # sign in front of the number $FA, the assembler does not interpret it as a literal number. Instead, $FA is interpreted as a memory address, which is what a number has to be in assembly language if it is not designated as a literal number and carries no other identifying labels.

Incidentally, if you did want the assembler to interpret $FA as a literal number, you would have to write it as #$FA. When # and $ both appear before a number, it is interpreted as a literal hexadecimal number. If the third line of the program was STA #$FA, however, it would result in a syntax error. That's because STA ("store the contents of the accumulator") is an instruction that has to be followed by a value that can be interpreted as a memory address—not by a literal number.

Processor Status Register

The processor status (P) register is built differently from the other registers in the 8502, and is used differently, too. It isn't designed for storing or processing ordinary 8-bit numbers, which the 8502's other registers are designed to do. Instead, its bits are used as flags that keep track of several kinds of important information.

Four of the status register's eight bits are called status flags. These four flags, and their abbreviations, are:

- carry (C) flag
- overflow (V) flag
- negative (N) flag
- zero (Z) flag

These four flags are used to keep track of the results of operations being carried out by the other registers inside the 8502 processor.

Because the P register is an 8-bit register, it has four more bits that could be used as flags, but only three of those flags are used. Called condition flags, they are used to determine whether certain conditions exist in a program. The P register's three condition flags are:

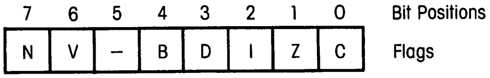
- interrupt disable (I) flag
- break (B) flag
- decimal mode (D) flag

The P register has one more bit—bit 5—which is not used.

Layout of the Processor Status Register

As illustrated in figure 3-3, the processor status register can be visualized as a rectangular box containing eight compartments. Each compartment in the box is one of the P register's eight bits, and each of these bits is used as a flag. If a bit has the binary value 1, then the bit is set. If it has the binary value 0, the bit is clear.

Figure 3-3
8502 processor status register



7	6	5	4	3	2	1	0	Bit Positions
N	V	–	B	D	I	Z	C	Flags

The bits in the 8502 status register—like the bits in all 8-bit registers—are customarily numbered from 0 to 7. By convention, the rightmost bit in an 8-bit register is bit 0, and the leftmost bit is bit 7. The positions of each bit in the 8502 status register can be seen in figure 3-3.

A Closer Look

Now let's take a closer look at each bit, or flag, in the processor status register.

Bit 0: Carry (C) Flag—As pointed out in chapter 2, it isn't easy to do *multiprecision* (multibyte) arithmetic using an 8-bit chip like the 8502. When the 8502 chip is required to perform an arithmetical operation involving a number greater than 255—or even if the *result* of a calculation is greater than 255—each number greater than 255 must be broken down into smaller numbers. Then, when the calculation is completed, all of the numbers that have been split must be joined before they can be output in a form that makes sense to the user.

This kind of mathematical cutting and pasting, as you can probably imagine, involves a lot of carrying (if the 8502 is performing addition problems) and borrowing (if the 8502 is performing subtraction). The carry (C) flag of the 8502 P register keeps up with all of this carrying and borrowing.

It is therefore considered good programming practice to clear the carry flag prior to an addition operation and to set the carry flag prior to a subtraction operation. If you don't clear the carry flag before every addition operation and set it before every subtraction operation, your calculations may be incorrect because of leftover results of previous calculations. The assembly language instruction to clear the P register's carry bit is CLC, which stands for "clear carry," and the instruction to set the carry bit is SEC, which stands for "set carry."

Here's how the carry bit works in 6502/8502 addition and subtraction operations. Before a multiprecision addition problem is performed in 6502/8502 assembly language, the carry flag of the P register is customarily cleared using the assembly language mnemonic CLC. Then the 8502 chip adds the lowest-order bytes of the two numbers being added. If this operation results in a carry to a higher-order byte, the carry flag is automatically set by the 8502 chip. Then, as higher-order bytes are added, the chip automatically uses the state of the carry flag to ensure that all carry operations are performed properly.

Because it is recommended that the carry flag be cleared before performing any addition operation, the ADDNRS.S program in listing 3-1 can be improved. The new version of the program is shown in listing 3-2. Preceding the addition operation with the CLC instruction clears the carry bit, ensuring that no unwanted carry is included in the operation. We'll see many more examples of how the carry bit works in addition problems later in this volume.

Listing 3-2
ADDNRS.S program, version 2

```
CLC
LDA #2
ADC #2
STA $FA
```

The carry flag is also used in subtraction problems. Before a subtraction operation is carried out, the carry bit is usually *set,* using the assembly language instruction SEC. Then, if the subtraction operation requires that a lower-order byte borrow a number from a higher-order byte, the number that is needed can be provided by the carry bit. The carry flag also has a number of other uses, and many of these uses are described in later chapters.

Bit 1: Zero (Z) Flag—When the result of an arithmetical or logical operation is zero, the status register's zero (Z) flag is automatically set. Addition, subtraction, and logical operations can change the status of the zero flag. For example, the zero flag is often tested to see whether two numbers are equal, particularly in programming loops that count down to zero.

When you write routines that use the zero flag, it's important to remember one 6502 convention that may seem odd at first: When the result of an operation is zero, the zero flag is set (to 1), and when the result of an operation is not zero, the zero flag is cleared (to 0). This convention is easy to forget and can trip you up if you aren't careful. There are no assembly language instructions to clear or set the zero flag. It's strictly a "read" bit, so instructions to write to it are not provided.

Bit 2: Interrupt Disable (I) Flag—Many Commodore programs contain *interrupts*—instructions that halt all 6502/8502 operations temporarily so that other, more time critical operations can take place. Some interrupts are called *maskable interrupts* because you can prevent them from taking place by setting the interrupt disable (I) flag of the processor status register. Other interrupts are called *nonmaskable* because they are essential to the operation of a computer and you can't stop them from taking place.

The most common reason for using the P register's interrupt disable flag is to write a sequence of code that will not work properly if an interrupt occurs while the code is executed. For example, if a program is setting up an interrupt and gets cut off in midstream by another interrupt, the whole program might crash. The best way to keep this type of disaster from happening is to set the interrupt disable flag, execute the sensitive segment of code, and then clear the interrupt disable flag. In this way, an unexpected interrupt can't occur and crash the program.

The assembly language instruction to clear the interrupt flag is CLI. The instruction to set the interrupt flag is SEI. Examples showing how this flag works are presented in later chapters.

Bit 3: Decimal Mode (D) Flag—The 8502 processor normally operates in *binary mode,* using standard binary numbers. But the chip can also operate in *binary coded decimal (BCD)* mode. To put your computer into BCD mode, you have to set the decimal mode (D) flag of the 8502 status register.

When the 8502 chip is put in BCD mode, it uses the same ten digits that are used in the standard decimal system: the numbers 0 through 9. The hexadecimal digits A through F are not used in the BCD system, and therefore are not recognized by the 8502 chip when the C-128 is in BCD mode. Table 3-1 shows how the C-128 translates the decimal numbers 1 through 15 into BCD, hexadecimal, and binary numbers when the BCD flag is set and the 8502 is in BCD mode.

Table 3-1 Decimal Conversion Chart for BCD Mode

Decimal	BCD	Hexadecimal	Binary
1	0001	1	0001
2	0010	2	0010
3	0011	3	0011
4	0100	4	0100
5	0101	5	0101
6	0110	6	0110
7	0111	7	0111
8	1000	8	1000
9	1001	9	1001
10	0001 0000	A	1010
11	0001 0001	B	1011
12	0001 0010	C	1100
13	0001 0011	D	1101
14	0001 0100	E	1110
15	0001 0101	F	1111

As table 3-1 shows, the binary numbers 1010 through 1111 (represented as A through F in the hexadecimal system) are not used in BCD notation. Instead, the decimal numbers 10 through 15 (expressed as A through F in the hexadecimal system) are represented in the BCD system by the decimal numbers 10 through 15, just as they are in the standard decimal system.

In other words, in BCD notation, 10 is written as 1 (0001 in binary) and 0 (0000 in binary), 11 is written as 1 and 1, 12 is written as 1 and 2, and so on. So, when the 8502 is in BCD mode, it converts the decimal values 10 through 15 into the binary numbers 0001 0000 through 0001 0101.

Because the binary numbers 1010 through 1111 are not used in the BCD system, it takes more memory to store numbers using BCD notation than it does to store numbers using non-BCD notation. In fact, in many applications—for example, in floating-point arithmetical operations—a full byte of memory is used for each decimal digit in a BCD number. And when BCD notation is used in this way, BCD numbers require even more memory.

Figure 3-4 shows how the decimal number 255 is stored in memory as a BCD number if each digit in the number is expressed as an individual byte. In comparison, figure 3-5 shows how the 8502 chip stores the decimal number 255 in memory if the BCD flag is not set.

Figure 3-4
Storing a number in BCD mode

Decimal number:	255		
BCD number:	2	5	5
Binary equivalent:	00000010	00000101	00000101

Figure 3-5
Storing a number in binary mode

Decimal number:	255	
Hexadecimal equivalent:	F	F
Binary equivalent:	1111	1111

As figures 3-4 and 3-5 illustrate, at the rate of one byte per digit, it takes three times as many bytes to store the number 255 in BCD notation as it does in binary notation. There are many applications in which BCD numbers use even more memory. For example, when the 8502 performs *floating-point arithmetic,* extra bytes are usually required to indicate how many digits there are in the number, whether the number is positive or negative, and how many decimal places there are in the number.

In floating-point arithmetic, which is often used in "number-crunching" operations because of its high degree of accuracy, it could take six or more binary numbers to express a three-digit decimal number. Figure 3-6 shows how the number 2.55 might be expressed as a 6-byte BCD number.

Figure 3-6 is only one illustration of how a number can be expressed as a BCD number in floating-point applications. There are many other methods for converting decimal numbers into BCD numbers for use in floating-point operations.

In addition to using extra memory, BCD arithmetic is slower than binary arithmetic. But because BCD numbers are based on 10, like conventional decimal numbers, they are more accurate in arithmetical operations that use fractions and decimal values. So BCD arithmetic is often used in programs when accuracy of calculations is more important than speed or memory efficiency.

Figure 3-6 Floating-point binary number

Decimal number: 2.55
Floating-Point BCD: 0011 0010 0000 0010 0101 0101

Meaning of Each BCD Digit

0011 The number has three digits
0010 Decimal point is to the left of the second digit
0000 The number is positive (0001 would mean a negative number)
0010 First digit (2)
0101 Second digit (5)
0101 Third digit (5)

Another advantage of BCD numbers is that it's easier to convert from BCD to decimal than it is to convert from standard binary to decimal. So BCD numbers are sometimes used in programs that require the instant display of numbers on a video monitor.

BCD numbers are discussed in more detail in chapter 9. For now, it's sufficient to note that when the status register's decimal mode flag is set, the 8502 chip performs all arithmetical operations using BCD numbers. You probably won't be using much BCD arithmetic in your assembly language programs—at least not for a while—so you'll usually want to make sure that the decimal flag is clear before the computer performs arithmetical operations.

The assembly language instruction that clears the decimal flag is CLD. The instruction that sets it is SED. The CLD instruction is often used before arithmetical operations take place to ensure that the 6502/8502 chip has not been placed and left in decimal mode. So a further improved version of the ADDNRS.S program might look like the one illustrated in listing 3-3.

Listing 3-3 ADDNRS.S program, version 3

```
CLD
CLC
LDA #2
ADC #2
STA $FA
```

Bit 4: Break (B) Flag—The break (B) flag is set automatically when the assembly language instruction BRK is used to halt a program. It is also set when certain error conditions stop a program. When a break occurs in a program, the break flag is set and certain error-checking operations take place. Because a BRK instruction sets the break flag, program designers often use BRK instructions during the debugging phase of writing programs. After the debugging of a program has been completed, any BRK instructions placed in the program for use during debugging are usually removed.

Other than the BRK instruction, there are no specific assembly language instructions to set or clear the break flag. Additional details on the operation of the BRK instruction and the break flag are in appendix A.

Bit 5: Unused Bit—For some reason, the programmers who designed the 8502 processor status register decided not to use one flag. This is the one.

Bit 6: Overflow (V) Flag—The overflow (V) flag detects an overflow from bit 6 to bit 7 in a binary number. The overflow flag is used primarily in addition and subtraction problems involving signed numbers. When the 8502 microprocessor performs calculations on signed numbers, each number is expressed as a 7-bit value, with the leftmost bit designating its sign. When bit 7 is used in this way, an overflow from bit 6 to bit 7 can make the result of a calculation incorrect. So, after a calculation involving signed numbers is performed, the V flag is often tested to see whether such an overflow has occurred. Then, if an unwanted overflow has occurred, corrective action can be taken. (More information on how the V flag is used in signed number operations is provided in chapter 9, which is devoted to 8502 arithmetic.)

The assembly language instruction that clears the overflow flag is CLV. The V flag is a read-only bit, so there is no specific instruction to set it.

Bit 7: Negative (N) Flag—The negative (N) flag is set when the result of an operation is negative and cleared when the result of an operation is zero. The negative flag is often used in operations involving signed numbers. In addition, the negative flag is often used to detect whether a counter in a loop has decremented past zero, and it is sometimes tested to see whether one number is less than another number. The negative flag has other uses that are discussed in later chapters. There are no instructions to set or clear the negative flag; it's strictly a read-only bit.

4

Writing an Assembly Language Program

Using three popular assemblers and the C-128 monitor

There are four ways to write a Commodore 128 assembly language program:

- You can use a large mainframe computer or minicomputer that is equipped with a professional software development system. We won't explore this method because chances are you don't have this kind of system.
- You can type and assemble the program using the miniassembler built into the C-128 machine language monitor.
- You can type and assemble the program while the C-128 is in C-64 mode, using an assembler designed for the Commodore 64. Then you can switch your computer to C-128 mode and run the program. This method is covered because some C-128 owners have moved up from the Commodore 64 and own C-64 assemblers.
- You can type and assemble the program using an assembler designed especially for the Commodore 128. For most C-128 owners, this is the best method for writing a C-128 assembly language program.

A Programmer's Toolkit

In this chapter, we will cover all of the methods of creating machine language programs for the Commodore 128. To write and assemble the programs in this chapter, we will use the following:

- C-128's built-in machine language monitor
- Merlin 128 assembler/editor system, which was designed specifically for use with the Commodore 128 and is manufactured by Roger Wagner Publishing, Inc., of Santee, CA
- TSDS (Total Software Development System) assembler/editor, which works with both the C-64 and the C-128 and is produced by the NoSync software company in Port Coquitlam, British Columbia
- Commodore 64 Macro Assembler Development System, which was originally designed for the C-64 and is manufactured by Commodore

To demonstrate how each of these tools can be used to produce a machine language program, we'll use the ADDNRS.S program, which was introduced in chapter 3. The first version of the program that we'll look at was created using a Merlin 128 assembler. Later we'll see how the program looks when it's written using a TSDS assembler, a Commodore 64 Macro Assembler, and the miniassembler built into the C-128 machine language monitor. No matter which assembler you're using, I suggest that you read the first part of this chapter, which covers the Merlin 128 assembler, because it's the only section that contains a full line-by-line explanation of how the program works. Then, if you like, you can skip ahead to any section that interests you.

What's an Assembler?

Before we start examining the Merlin 128 assembler, let's get one potentially confusing topic out of the way: As you may have concluded by now, the word *assembler* can have different meanings, depending upon the context in which it is used. When programmers speak of an assembler, they're sometimes talking about one part of a software development package—the part that does the actual work of converting assembly language into machine language. But the word *assembler* can also refer to a complete assembly language programming package, such as the Commodore 64 Macro Assembler Development System or the Merlin 128 assembler/editor program. And software packages like these usually include more than just an assembler. Other types of programs that are often contained in assembler software packages include editors, monitors, loaders, and debugging utilities.

Merlin 128 Assembler

Now we're ready to take a look at the Merlin 128 assembler. The Merlin 128 assembler/editor system contains a number of programs, all stored on a single disk. These programs are divided into five modules:

- Executive module
- Editor module
- Assembler module
- Monitor module
- Symbol Table Generator module

When you use the Merlin 128 assembler, the modules that you'll encounter most often are the Executive, Editor, Assembler, and Monitor modules. You'll seldom have to worry about the Symbol Table Generator module. Its job is to compile tables of constants and variables, and it does its work automatically and quite transparently, usually without any assistance from the programmer.

The Merlin 128 assembler/editor, like most commercial programs for the C-128, boots automatically. So, to get the assembler up and running, put the Merlin 128 disk in your disk drive and either press the C-128's reset button or turn the system on. Then, after a few moments of disk spinning, Merlin's master menu will appear on your screen.

Merlin is a menu-driven assembler, so the menu that appears when you boot the Merlin disk is what you'll use to select the assembler's functions. When Merlin's master menu is on the screen, the assembler is in Executive mode. The Merlin 128 module that controls this mode is the Executive module.

Merlin's Menu

Figure 4-1 shows what the menu looks like when Merlin is in Executive mode.

Figure 4-1 Merlin's menu screen

```
              Merlin's Menu

C :Catalog
L :Load source
S :Save source
A :Append file
N :New source
R :Read text file
W :Write text file
D :Drive change
E :Enter ED/ASM
O :Save object code
G :Run program
X :Disk command
M :Monitor
B :Basic
%
```

All of the menu options shown in figure 4-1 are explained in detail in the user's manual that comes with the Merlin 128. So there's no need to present a long explanation of each menu option in this chapter. For now, it's sufficient to note that Merlin can do quite a few things in Executive mode, from loading and saving source code and object code to listing the contents of a disk (using the menu's C command). You can read and write text files using Merlin's R and W menu commands. You can even format disks, scratch (erase) files from disks, and perform numerous other disk-management functions using the Executive menu's X command.

To put Merlin into Editor mode—the mode you'll be using to write assembly language programs—select choice E from the Executive menu as follows. At the bottom of your screen, just below the menu, you should see a % sign followed by a flashing cursor. The % prompt is the prompt you'll always see when Merlin is in Executive mode. When the assembler is in Editor mode, the prompt changes to a : prompt, and when it's in monitor mode, the prompt is the $ symbol.

When you've located the % prompt, type:

```
E
```

for "enter Editor/Assembler mode." As soon as you type an E and press Return, Merlin's Editor module goes into action. To let you know that it's in edit mode, Merlin clears the screen and prints the word "Editor" at the top of the screen, followed by a : prompt and an underline cursor. When all this has taken place, type:

```
A
```

for "append." Then you'll be ready to start writing a program using the Merlin 128 system.

If you're familiar with the Merlin 64 assembler for the Commodore 64 or any versions of Merlin assemblers for Apple II computers, you will probably notice at this point that the Merlin 128 is considerably different from all of its predecessors. All previous Merlin assemblers have line-oriented editors—screen editors that, like most BASIC editors, require the user to edit programs line by line. But the Merlin 128 has a full-screen editor—an editor that allows you to move the cursor in any direction and make changes anywhere on the screen, similar to the way text is edited using a word processor.

When you type the A command to start editing a program, Merlin clears the screen again and places the cursor at its "home" position, in the upper left corner of the screen. The cursor you see there is an I displayed in reverse video. This cursor means that Merlin is in insert mode. When you type a character in insert mode, previously typed characters move to the right (to accommodate the new character). As we'll see later, Merlin also has a typeover mode. When you type a character in typeover mode, the new character replaces the character under the cursor. When Merlin is in typeover mode, the cursor changes to a solid rectangle (without the reverse I).

When Merlin's screen editor goes into action, the number 1 appears in a window in the upper right corner of the screen. That number is a line counter, and when the counter is at 1, it means Merlin is waiting for you to write the first line—or line 1—of an assembly language program. After you write line 1 and press Return, the number in the counter window changes to 2, which means Merlin is ready to accept the second line. As you continue to write lines of code, Merlin increments the line counter, unless you move the cursor to a previously written line. In that case, Merlin decrements its line counter, so the counter always displays the number of the line that contains the cursor.

When you write a program using the Merlin assembler, you can move the cursor to any point in the program using the C-128's cursor (arrow) keys. You can also toggle between insert mode and typeover mode by typing Control-I.

To delete a line, press D while holding the Commodore key (C=). You can create the space needed to insert a new line in a program by pressing the Return key while Merlin is in insert mode.

ADDNRS.S Program

Merlin offers many other editing features, including functions to delete blocks of code, move blocks of code, and copy blocks of code from one part of a program to another. Merlin's editing functions are described in detail in the instruction manual that comes with the assembler. But you don't need to know all of Merlin's editing functions to write an assembly language program. In fact, using only the instructions we have just covered, you can write an assembly language program now. Start by simply typing an asterisk, as follows:

```
*
```

and pressing Return. The cursor moves to the next line, and the number in Merlin's line counter window automatically advances to 2.

Now, without moving the cursor, type:

```
* ADDNRS.S
```

and press Return. Then, when Merlin advances to line 3, type another asterisk.

This is what you should see on your screen now:

```
*
* ADDNRS.S
*
```

Continue typing until you've entered the program in listing 4-1.

Listing 4-1
ADDNRS.S program

```
*
* ADDNRS.S
*
        ORG   $1300
ADDNRS  CLD
        CLC
        LDA   #2
        ADC   #2
        STA   $0C00
        RTS
        END
```

As you type the ADDNRS.S program, you'll probably notice that Merlin tabulates columns automatically, dividing a program neatly into easy-to-read fields. Later in this chapter, we'll discuss program listings at greater length. You can find more on the subject in the Merlin 128 instruction manual.

When you have typed the last line in the ADDNRS.S program, you can leave Merlin's editing mode by simultaneously pressing the

Commodore key and the white left-arrow key (not the grey left-arrow key that moves the cursor). Then Merlin will list your program, complete with line numbers, on the computer screen.

Because the line numbers generated by Merlin always start with 1 and progress in increments of 1, they can—and do—change dynamically while you write a program. Therefore, they are sometimes referred to as *relative* line numbers. And, although the current line number always appears in the line counter window on the screen, the only way that you can see the line numbers in a program is to use the L (for "list") command, which lists the program on the screen.

When you have listed a program, Merlin displays another : prompt, and you can put the assembler back into editor mode by typing another A command. Alternatively, you can put Merlin back into editor mode by typing the E command, followed by a line number. When Merlin goes into editor mode in response to an E command, the portion of the program that contains the desired line appears on the screen, and the cursor is positioned at the beginning of the line to be edited.

You can also return Merlin to editor mode by typing I (for "insert") followed by the number of the line to be inserted. The I command works like the E command, except it creates a blank line in the program at the point where the new line is to be inserted.

When a program has been listed and the : prompt appears on the screen, you can delete a line by typing D (for "delete"), followed by the number of the line (or lines) you want to delete. Suppose you want to delete lines 2 and 3 in the preceding listing. Type:

```
D2,3
```

after the : prompt. Then restore the lines you've deleted by using the A command.

Here's an important point to remember about the Merlin 128 assembler. When a block of code is inserted into a program being written on a Merlin assembler, or when a block is deleted, Merlin automatically adjusts all subsequent line numbers to accommodate the change. So, when you write a program using the Merlin 128, remember that the line numbers in the program—particularly the line numbers that follow material being inserted, deleted, or edited—can always change without notice.

In addition to the A, I, and D commands, Merlin also has commands to copy lines, move lines, find and replace strings, and perform many other useful functions. You can find full details on how to use all of these functions in the Merlin 128 instruction manual.

We've examined the ADDNRS.S program so many times that it's probably beginning to look familiar. The Merlin 128 version of the program, like the version of the program presented in chapter 3, adds the literal numbers 2 and 2 and then stores their sum in memory. This all happens in lines 7, 8, and 9.

As familiar as all of this is, however, the Merlin version of the program does include one or two features that we haven't previously encountered. These new features include the asterisks in the first three lines of the program, the ORG directive in line 4, and the END directive in line 11.

Listing 4-2 is a diagram that "spaces out" the ADDDNRS.S program so you can get a clearer picture of how it's written. In listing 4-2, the program is divided into five vertical columns, and each column has a heading that describes the kind of information it contains.

Listing 4-2 Expanded ADDNRS.S program with headings

Line Number	Label	Op Code	Operand	Comments
1	*			
2	* ADDNRS			
3	*			
4		ORG	$1300	
5	ADDNRS	CLD		
6		CLC		
7		LDA	#2	
8		ADC	#2	
9		STA	$0C00	
10		RTS		
11		END		

Line Numbers in Assembly Language Programs

At this point, it might be helpful to pause for a brief discussion of how line numbers are used in assembly language programs. Line numbers are not an essential part of an assembly language program because an assembly language program, unlike a BASIC program, never uses a line number to refer to a jump address or to call a subroutine. If line numbers are used in an assembly language program, they are placed in the program only as a convenience to the programmer.

Not all assemblers handle line numbers in the same way. For example, the Commodore Macro Assembler lets you assign your own line numbers, and the TSDS assembler lets you decide whether you want to assign line numbers or let your assembler do it for you. And some assemblers, such as the ORCA/M assembler for the Apple II, are equipped with full-screen editors that don't use line numbers at all.

Fields in Assembly Language Programs

In assembly language jargon, four of the five columns shown in listing 4-2 are known as *fields.* The column that is not considered a field is the column labeled Line Number. As mentioned, line numbers are optional in assembly language programs because assembly language routines and subroutines are never referred to in a program by their line numbers; when a machine language program jumps or branches to a routine or a subroutine, the routine or subroutine is referred to not by its line number but by a defining *label*—which *is* a valid field in an assembly language program. The second column in listing 4-2 is the label column.

Label Field

Although labels have a field of their own, they are also optional in 6502/8502 assembly language programs. Some assemblers, such as the miniassembler built into the Commodore 128 machine language monitor, are not designed to handle labels—and that is a significant shortcoming of the C-128 monitor. When labels are not used in an assembly language program, the only way to access a routine or a subroutine is to call it using the actual memory address at which it begins. And memory addresses of routines and subroutines in a program tend to change quite often during the writing and editing of the program.

If you can access a routine or a subroutine with a label, all of this shifting of memory addresses is quite invisible; after a routine is assigned a label, an assembler that recognizes labels can always locate it without difficulty, no matter how many times the starting address changes. But when a program is written using an assembler that does not recognize labels, the life of the programmer is more difficult; each time the starting address of a routine changes, every reference to that routine anywhere in the program must also be changed—by hand.

Although the miniassembler built into the C-128 monitor doesn't recognize labels, most full-featured assembler/editor packages do. The Merlin 128 assembler, the TSDS assembler, and the Commodore Macro Assembler System all recognize labels, and so do most other commercially available assembler/editor programs.

When labels are used in an assembly language program, they always occupy the first field in the program listing. Thus, when you write a program with an assembler that uses line numbers, the label field always appears immediately after the line number. In the ADDNRS.S program listed in listing 4-2, the abbreviation ADDNRS in line 5 is a label, and thus appears in the first field, or second column.

Because the ADDNRS.S program is identified with a label, the whole program could be used as a subroutine in another program. If ADDNRS.S was used as a subroutine in another program, and if the other program was assembled using an assembler that recognized labels, then the ADDNRS.S program could be accessed using the statement JSR ADDNRS (which is equivalent to GOSUB in BASIC) or the instruction JMP ADDNRS (which works like BASIC's GOTO instruction). Also, if ADDNRS was used as a subroutine, the RTS (return from subroutine) instruction in line 10 would end the subroutine and return control to the main program. (The instructions JSR and JMP are discussed at greater length in later chapters.)

A label can be as short as one character and as long as the assembler being used permits. Most assembly language programmers write labels that contain three to six (or sometimes eight) characters.

Op Code Field

An operation code (or op code) mnemonic is just a fancy name for an assembly language instruction. There are 56 op code mnemonics in the 6502/8502 instruction set, and they are the only ones that can be used in Commodore 128 assembly language instructions.

In source code listings of assembly language programs, op code mnemonics, such as CLC, CLD, LDA, ADC, STA, and RTS, are typed in the op code field. When you write a program using the Merlin 128 assembler, each op code mnemonic must start at least two spaces after a line number, or one space after a label. An op code mnemonic placed in the wrong field will not be flagged as an error when you type your program, but will be flagged as an error when your program is assembled.

The op code field in a source code listing is also used for *directives*, or *pseudo ops*—words and symbols that are entered into a program like mnemonics but are not officially included in the 6502/8502 instruction set. The main difference between an op code and a pseudo op is that an op code tells a program what to do, and a pseudo op tells an assembler what to do. Pseudo ops, unlike 6502/8502 op code mnemonics, vary from assembler to assembler. So, before you use a pseudo op in an assembly language program, it's important to find out whether the assembler you're using recognizes the pseudo op.

In the ADDNRS.S program, the ORG abbreviation (line 4) and the END statement (line 11) are directives, or pseudo ops, that are recognized by the Merlin 128 assembler. In programs written using the Merlin 128, the ORG directive tells Merlin where an assembly language program is to be stored in memory after it is assembled. The END directive tells Merlin where to stop assembling and end the program.

Operand Field

The operand field in a Merlin 128 assembler program starts one space (or one tab) after the mnemonic field. Some mnemonics require operands, while others don't. Instructions we have encountered so far that do not require operands include CLC, CLD, and RTS. Instructions that do require operands include LDA, STA, and ADC. The use of operands is covered in more detail in chapter 6, which focuses on the addressing modes used in 6502/8502 assembly language.

Comment Field

Comments in assembly language programs are like remarks in BASIC programs; they don't affect the execution of a program in any way, but are often extremely useful because they can help explain how each step in a program works and are thus an important part of the documentation of a program.

There are two ways to include a comment in source code listings written on the Merlin 128 assembler. One method is to precede the comment with an asterisk and put the comment in the label field of a listing. The other method is to precede the comment with a semicolon and put it in the comments field, which follows the operand field. This method of including comments is used in many of the programs in this volume.

Taking It Line by Line

Now that we've examined the ADDNRS.S program field by field, let's go over it again in more detail, line by line.

Lines 1 through 3: Comments

Lines 1 through 3 are comments. Line 2 explains what the program does, and lines 1 and 3 set off the explanatory line by printing asterisks.

Because it's often amazing how little of a program one remembers after the ink is dry, it's considered good programming practice—in assembly language, as in most other programming languages—to use remarks liberally. So comments have been used quite extensively in the programs in this volume.

Line 4: ORG $1300

Line 4 is the origin line of the ADDNRS.S program. Every program written using the Merlin 128 assembler must start with an origin line. As you may remember from chapter 1, when a computer runs a machine language program, the first thing it does is go to a predetermined memory location and take a look at the value stored at that address. So, when you write an assembly language program, the first thing you have to do is tell your assembler where to assemble the program in memory.

When you begin an assembly language program with an origin directive, Merlin sets an internal counter that it then uses to keep track of the instructions and data used in the program. This counter is called, logically enough, the program counter. Merlin's built-in program counter, like the program counter in the C-128's 6502/8502 processor, always contains the address of the next instruction to be used in whatever program the assembler is working on.

The origin directive looks simple enough to write; all it does, after all, is tell Merlin where in memory to assemble a program. But deciding where a program should reside in memory after it's assembled can be a difficult task, especially for the beginning assembly language programmer. You cannot store user-written programs in many of the C-128's memory blocks because these blocks are reserved for other uses—for example, to hold the computer's operating system, its built-in character set, and its BASIC 7.0 interpreter. Even the assembler/editor used to assemble a program takes up memory space, and if you try to execute the program while the assembler is in memory, the program might crash.

Deciding where to store a program in a computer's memory is a particularly tricky job when the memory layout of the computer is as complex as that of the Commodore 128. We won't be tackling this topic until we get to chapter 10, which is devoted to C-128 memory management. For now, it's sufficient to know that it's usually safe to store assembly language programs that are under 2K in a block of RAM that extends from memory address $1300 to memory address $1BFF. There's also a 512-byte block of usually free RAM that extends from memory address $0C00 to memory address $0DFF.

It isn't *always* safe to use these two blocks of memory because the segment that starts at $1300 is technically reserved for foreign language systems and function key software, and the block that starts at $0C00 is reserved for RS-232 serial port buffers. But if your program doesn't make use of a foreign language system and function key software, you can use these segments.

Now you know why line 4 in the ADDNRS.S program sets the assembler's program counter at $1300, and why line 9 stores the sum of 2 and 2 in memory address $0C00.

Line 5: ADDNRS CLD

The label field in line 5 of the ADDNRS.S program is used to name the routine ADDNRS. So, if we ever decide to use ADDNRS.S as a subroutine in a larger program, the subroutine will have a name. Then, as explained earlier in this chapter, we can call the subroutine by using its label, instead of its actual memory address. We also give the routine a label because a label can remind us of what a routine does (or, during the debugging phase of a program, what the routine is supposed to do).

Following the ADDNRS label in line 5 is the mnemonic CLD. We're using binary numbers in this program, not binary coded decimal (BCD) numbers, so the instruction CLD clears the decimal mode flag of the 8502 processor status register. You don't have to clear the decimal flag before every arithmetical operation in a program, but it's a good idea to clear it before the first addition or subtraction operation because it may have been set during a previous program.

Line 6: CLC

Line 6 contains the CLC ("clear carry") statement. The status register's carry flag is affected by so many kinds of operations that it's considered good programming practice to clear it before every addition operation and set it before every subtraction operation. It takes only about half a millionth of a second, and just one byte of RAM—and compared to the time and energy that debugging can cost, that's a bargain.

Line 7: LDA #2

Line 7, LDA #2, is a very straightforward instruction, and we already know what it means ("load the accumulator with the number 2"). The first step in an addition operation is always loading the accumulator with one of the numbers that is to be added. The # sign before the number 2 means that it's a literal number, not an address. If the instruction was LDA 2, then the accumulator would be loaded with the contents of memory address 0002, not the number 2.

Line 8: ADC #2

ADC #2, in line 8, is also a straightforward instruction. It means that the literal number 2 is to be added to the number that's in the accumulator (in this case, another 2). As mentioned, there is no 6510 assembly language instruction that means "add without carry." So the only way that an addition operation can be performed without a carry is to clear the status register's carry flag and then perform an "add with carry" operation.

Line 9: STA $0C00

Line 9, STA $0C00, completes our addition operation. It stores the contents of the accumulator in memory address $0C00. Note that the # symbol is not used before the operand ($0C00) in this instruction because in this case the operand is a memory address, not a literal number.

Line 10: RTS

Line 10 contains the RTS mnemonic. If RTS is used at the end of a subroutine, it works like the RETURN instruction in BASIC; it ends the subroutine and returns to the main body of the program, beginning at the line following the RTS instruction.

But if RTS is used at the end of the main body of a program—as it is here—it has a different function. Instead of passing control of the program to a different line, it ends the whole program and returns control of the computer to the input device that was in control before the program began—usually a plug-in ROM cartridge, a disk operating system (DOS), a keyboard screen editor, or a machine language monitor.

Line 11: END

Just as the ORG directive begins an assembly language program, the END directive (line 11) ends it. The END directive tells the assembler to stop assembling, and that's exactly what the assembler does—even if there's more source code after the END directive. Therefore, the END directive can be a powerful debugging tool. You can put the

END directive wherever you want in a program you're debugging, and that's where the assembler will always stop assembling (until you remove the directive).

When you've finished debugging your program, you can use the END directive to end the program neatly. Before that, of course, you must remove any leftover END directives—that is, if you want your final program assembled. When you've finished debugging and your program is finished, the program should contain only one END directive, where it belongs—at the very end of your program.

Printing the Program

When you've finished typing your source code listing using the Merlin editor, you can print it by typing the command:

```
PRTR 4
```

if your printer is installed as device number 4. If not, use the appropriate device number. After you type the PRTR command, followed by the device number of the printer, type LIST to print your program.

Assembling the Program

To assemble the ADDNRS.S program using the Merlin assembler, simply type the ASM command after the : prompt. Merlin will ask if you want to update your source code file—with the current date, for example. If you don't want to update your file, you can type N (for "no") and Merlin will assemble your source code program *very* rapidly!

In just a moment, we'll save the ADDNRS.S program on a disk. First, though, let's take time out to compare the object code (which the assembler has generated) with the source code (from which the object code was derived). Two listings of the program are shown in listing 4-3: a source code listing and an object code listing, which the source code produces when it's run through an assembler. The meaning of each line of code is provided in the right-hand column.

Saving the Program

Now we'll save both your source code listing and object code listing on a disk. First, type Q (for "quit") after the : prompt to return the assembler to Executive (menu) mode. Merlin's main menu will reappear. Then, save the source code by selecting menu choice S and save the object code with menu choice O.

(The Merlin menu also offers a choice labeled W, for "write text file." When you choose that menu selection, the assembler saves

Listing 4-3 Source code and object code compared

Source Code	Object Code	Meaning (Do not type)
`CLD`	`D8`	Clear status register's decimal mode flag
`CLC`	`18`	Clear status register's carry flag
`LDA #2`	`A9 02`	Load accumulator with the number 2
`ADC #2`	`69 02`	Add 2, with carry
`STA $0C00`	`8D 00 0C`	Store result in memory address $0C00
`RTS`	`60`	Return from subroutine

your object code listing on a disk as an ASCII text file, not as an executable binary file. You can't run a text file on your computer as you can a binary file, but listings of machine language programs in ASCII format do have their uses. For example, you can print a text file on paper. You can also transmit a text file from computer to computer over a telephone line; then the recipient can convert the program into a binary file and run it.)

When you type S, O, or W to save a source code or object code listing, Merlin asks you what you'd like to name your program. When you name the program, you don't have to add any special suffix to indicate whether it's a source code listing or an object code listing because Merlin does it automatically. If you're saving a source code listing, the assembler automatically adds an S suffix to your file name. If you're saving an object code listing, Merlin automatically appends an O suffix to the file name.

The following paragraphs are for owners of the Commodore 64 Macro Assembler Development System, manufactured by Commodore. If you don't own a Commodore Macro Assembler system and don't care how it works, you can skip to the next section, which is about the NoSync TSDS assembler. If you don't care about the TSDS assembler either, you can move to the end of this chapter, where you will find a short discussion of the built-in C-128 machine language monitor.

Commodore 64 Macro Assembler

The Commodore 64 Macro Assembler Development System, like the Merlin 128 assembler/editor package, comes on a single disk but includes a number of programs. These programs are:

- An assembly language editor called EDITOR64. An assembly language editor, as we have seen, is a program that can be used to write assembly language programs.
- An assembler, which is used to convert source code to object

code. The assembler included in the Commodore 64 Macro Assembler Development System is called ASSEMBLER64.

- A loader. When an assembly language program is converted into object code using the Commodore 64 assembler, the listing that is produced is not a true machine language listing, but an ASCII listing that must subsequently be converted into machine code. The job of the Commodore 64 loader is to produce this code from the pseudo object code generated by the Commodore assembler.

  Actually, there are two loader programs in the Commodore 64 assembler package. These programs, called LOLOADER64 and HILOADER64, perform identical functions but are placed in different parts of RAM when they are loaded into the computer's memory. If one of the loaders will overwrite machine code that you've written, you can use the other loader. The LOLOADER program goes into memory beginning at address $0800, and HILOADER starts at RAM location $C800.
- A machine language monitor. When the Commodore 64 assembler was designed, there wasn't a Commodore 128, so there also wasn't a built-in C-128 machine language monitor. A monitor was therefore included as part of the Commodore 64 Macro Assembler System. The monitor that comes with the Commodore 64 assembler is similar to the C-128 monitor, and is designed to be used for the same purposes. Some of those purposes are described in chapter 1, and others are covered later in this chapter.

Two Monitors, Too

There are actually two monitors in the Commodore 64 assembler package, just as there are two loaders. And again, the only difference between them is where they reside in the computer's memory when they're loaded. One goes into RAM beginning at memory address $8000, and the other starts at $C000.

If you still don't quite understand all of these references to memory addresses, don't worry about it too much. For now, it's sufficient to know that the ADDNRS.SRC program, because it starts at memory address $1300, is compatible with either of the monitors that come with the Commodore 64 assembler—as well as with the monitor that's built into your C-128.

DOS Wedge

In addition to all of the programs mentioned, the Commodore 64 assembler/editor disk also contains two useful programs called DOS WEDGE64 and BOOT ALL. If you do much programming using a Commodore 64, you probably know what a DOS wedge is; it's a utility that can come in handy when you're writing C-64 programs because it makes the C-64's disk operating system a little easier to use. So a DOS wedge also comes with the C-64 Macro Assembler System. Both the DOS wedge and the BOOT ALL program are discussed in more detail later in this chapter.

ADDNRS.SRC Program

First, though, here's a listing of the ADDNRS program—the same program presented in the first section of this chapter—written using the Commodore 64 assembler. To distinguish this latest version of the program from the version produced using Merlin, we'll call this new version ADDNRS.SRC rather than ADDNRS.S. Shortly, you'll get an opportunity to type, assemble, and run the ADDNRS.SRC program. A listing of the program appears in listing 4-4.

Listing 4-4 ADDNRS.SRC program

```
10 ;
20 ;ADDNRS.SRC
30 ;
40  *=$1300
50 ;
60 ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
100  STA $0C00
110  RTS
120  .END
```

It's not necessary to explain in detail again how the ADDNRS.SRC program works. Just as it did in its Merlin version, it adds 2 and 2, and then stores their sum in memory address $0C00 (3072 in decimal notation). In this version of the program, this happens in lines 80, 90, and 100.

Listing 4-5 is an expanded diagram of the program that may give you a clearer understanding of how it's written. This listing, like the expanded diagram used to illustrate the Merlin version of the program, is divided into five columns. And each column has a heading that describes the kind of information it contains.

Line Numbers Revisited

The Commodore 64 assembler, like most assemblers, uses line numbers. But they work more like BASIC line numbers than line numbers in the Merlin 128. And the Commodore assembler, unlike Merlin, doesn't assign line numbers automatically. When you use the Commodore assembler, you can either write your own line numbers or instruct the assembler to assign them automatically by using a special AUTO command. Instructions for using the AUTO command are

Listing 4-5 Expanded ADDNRS.SRC program with headings

Line Number	Label	Op Code	Operand	Comments
10	;			
20	;ADDNRS.SRC			
30	;			
40		*=$1300		
50	;			
60	ADDNRS	CLD		
70		CLC		
80		LDA	#2	
90		ADC	#2	
100		STA	$0C00	
110		RTS		
120		.END		

given on page 21 of the instruction manual that comes with the Commodore assembler.

As you can see, the line numbers in our sample ADDNRS.SRC program progress from 10 to 120 in increments of 10, just like the numbers in a typical BASIC program. They don't have to be written that way, but they usually are. When a program is written using the Commodore 64 assembler, line numbers are typed flush left, as they generally are in a BASIC program. So the line numbers in a Commodore 64 assembler listing occupy the first column in the program's source code listing.

Label Field

Labels always occupy the first official field in Commodore 64 assembler programs, as they do in programs written with the Merlin 128 assembler. Exactly one space—not two—must be left between a line number and any label that follows. If you start a label two or more spaces after a line number—or if you use the tab key to get to the label field—you may clobber your program.

Op Code and Operand Fields

Op code mnemonics and operands occupy the third and fourth fields of programs written with the Commodore 64 assembler, just as they do in Merlin 128 source code listings. But some of the op code directives used by the Commodore 64 assembler are different from those used in Merlin 128 programs. For example, look at line 40 in the ADDNRS.SRC program. Instead of using the abbreviation ORG to identify the origin line of a program, the Commodore assembler uses an asterisk followed by an equal sign. The standard format when using these symbols is shown in line 40 of the ADDNRS.SRC program:

```
*=$1300
```

The third and fourth fields of the ADDNRS.SRC program—like the third and fourth fields in the ADDNRS.S program presented earlier in this chapter—are used for op code mnemonics and operands. Some of the pseudo ops used in source code listings written with the Commodore assembler are slightly different from those used in Merlin listings. One such pseudo op is the .END directive in line 120 of the ADDNRS.SRC program; it is preceded with a period, which the Merlin END directive lacks.

Comment Field

There are also differences in the way in which you write comments using the Commodore and Merlin assemblers. In programs written with the Commodore assembler, a comment that begins in field 2 is preceded by a semicolon rather than an asterisk. A comment preceded by a semicolon can also appear in the section after the instruction fields (op code and operand fields). If you use the comment field at the end of a line and don't have room for the entire comment, you can continue your comment on the next line by simply typing a space, a semicolon, and the rest of your remark.

A line-by-line explanation of the ADDNRS program was provided earlier in this chapter, in the section on the Merlin 128 assembler. If you skipped that section because you're using the Commodore 64 assembler, it would be a good idea to back up and read the line-by-line program analysis now because the same explanations apply to the version of the program written on the Commodore 64 assembler. After you do that, you'll be ready to write and run the ADDNRS.SRC program.

Loading the Editor 64 Program

Ready? Then sit down at your computer, put it into C-64 mode, and slip your Commodore 64 Macro Assembler Development System disk into the disk drive. Then you can load the assembly language editor that's on the disk into your computer, or, if you prefer, you can start your editing session by loading a wedge program that's also on the disk. A wedge, in case you're not yet familiar with the term, is a machine language program that makes life with a Commodore 64 much simpler. When you're using a wedge program, the commands that perform DOS functions are greatly simplified. For example, when the red light on your disk drive starts blinking because of some saving or loading problem, the source of the trouble is easy to track down if you're using the Commodore assembler's wedge program. All you have to do is type the symbol @, and the wedge will provide you with an error message that tells you exactly what went wrong.

If you want to use the wedge program that's on the assembler/editor disk, load it by typing LOAD "DOS WEDGE64",8 followed by the RUN command. Then, instead of typing the line LOAD "EDITOR64",8,1 to load your Commodore 64 editor, you can type the line %:EDITOR64. Then you can put your editor program into operation with the command SYS49152.

If that all sounds too complicated to remember, you may be pleased to learn that there's a much simpler way to load both the DOS wedge program and the Commodore 64 editor program. Just insert the assembler/editor disk into the drive and type:

```
LOAD "BOOT ALL",8
```

When you've typed that line, press Return, wait for the READY prompt, and then type:

```
RUN
```

These two simple commands load the DOS WEDGE64 program, the EDITOR64 program, and the HILOADER64 program (more on that one later) into the computer's memory. And you don't have to type SYS49152 to run the EDITOR64 program; BOOT ALL takes care of that automatically. All you have to do is load BOOT ALL and start programming.

Unfortunately, however, there are some circumstances under which you can't use the BOOT ALL routine. For example, if you want to use the LOLOADER utility instead of the HILOADER routine (as we will be doing later in this chapter), you can't use BOOT ALL.

Storing Assembly Language Programs

When you have EDITOR64 up and running, it would probably be a good idea to put a formatted disk into the disk drive so that you can store the assembly language programs that you're almost ready to start writing. Many of the programs used in this book build on each other, so if you start saving them now, you can save yourself a lot of typing.

If you've saved any of the programs in preceding chapters on a disk, you can use that same disk for your assembly language programs. If you haven't started a program disk yet, now is a good time to put a blank (but formatted) program storage disk in your disk drive.

Using the C-64 Assembler/Editor

When your editor is up and running, you'll see COMMODORE 64 EDITOR and a READY prompt on the screen. You can then type the ADDNRS.SRC program. To save you some page flipping, another listing of the program appears in listing 4-6.

Listing 4-6
ADDNRS.SRC
program

```
10 ;
20 ; ADDNRS.SRC
30 ;
40  *=$1300
50 ;
60 ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
100  STA $0C00
110  RTS
120  .END
```

More Notes on Spacing

The Commodore assembler is very fussy about spacing, so you'll have to be careful when you type the ADDNRS.SRC source code listing. Here are a few helpful tips about typing spaces in assembly language programs written using the Commodore 64 assembler.

In the lines that contain semicolons, there should be only one space between the line number and the semicolon. In line 40, however, there should be two spaces between the line number and the asterisk because * is a directive and directives appear in the op code field of Commodore 64 assembler programs.

In line 60, there should be one space between the line number and the ADDNRS label, and one space between ADDNRS and the mnemonic CLD. In lines 70 through 110, there should be two spaces between each line number and the op code that follows. And in line 120, there should be two spaces between the line number and the .END directive.

If you make a mistake while typing a line, you can correct it in the usual Commodore 64 fashion, using the C-64 cursor control (arrow) keys. After you've typed the ADDNRS.SRC program into your computer, type the word LIST, and you should see a screen display that looks like the one shown in listing 4-7.

Listing 4-7
Listing the
ADDNRS.SRC
program

```
10 ;
20 ; ADDNRS.SRC
30 ;
40  *=$1300
50 ;
60  ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
100  STA $0C00
110  RTS
120  .END
```

Printing the Program

You can print the ADDNRS.SRC program in the same way that you print a C-64 BASIC program. Just type:

```
OPEN 1,4,4
```

The first number can be any value between 1 and 255 and the second value is the device number for a printer attached to a Commodore. If your printer is not device number 4, you will have to use a different device number. After your printer's device channel is open, you can type:

```
CMD 1
```

—or CMD followed by whatever optional number you've chosen. The CMD command routes the computer's output to the printer, instead of the screen. Then type the command:

```
LIST
```

to get a hardcopy printout of the ADDNRS.SRC source code listing. Easy enough, right? No sooner typed than finished! After you've printed the program, though, don't forget to type:

```
CLOSE 1
```

so that the screen will become your primary output device again.

Saving the Program

Now, if your listing looks all right, you can save your program on a disk. Make sure that your formatted program storage disk is in the disk drive, and that the drive is initialized. Then type:

```
PUT "ADDNRS.SRC"
```

The top red light on your disk drive should now go on, and the disk you're storing the program on should start to spin. When your disk drive's "busy" light goes off, the source code should be safely recorded on a disk under the file name:

```
ADDNRS.SRC
```

Are you sure, though, that your program has been stored safely? To find out, you can type:

```
NEW
```

Then type:

```
GET "ADDNRS.SRC"
```

Now type:

```
LIST
```

—and if you've succeeded in saving the program on disk, you'll see it listed on the screen. But you may notice that there's been a change in the program: The line numbers, instead of progressing from 10 to 120, will extend from 1000 through 1110!

And why did that happen? Well, your Commodore 64 assembler/editor "likes" programs better when they're numbered that way. This convention does make a certain amount of sense. If a routine starts with a line numbered 1000, it's easy to put other routines both before and after it. But if you don't want your source code to start with line 1000, it's easy to change; the Commodore 64 editor has a line renumbering utility that's very easy to use. For further details, consult your Commodore 64 editor/assembler instruction manual.

Assembling the Program

Now you know how to write, save, and load an assembly language source code listing using the Commodore 64 assembler. So you're ready to learn how to use the Commodore assembler to assemble a program.

If you're using the DOS wedge program, it's easy to load the assembler that's on the C-64 assembler/editor disk. Remove the user disk from the disk drive, insert the assembler/editor master disk, and type:

```
/ASSEMBLER64
```

(If you're not using the DOS wedge, you'll have to type LOAD "ASSEMBLER64",8—and if the assembler fails to load, you may never know why!)

When you've loaded the assembler, type:

```
RUN
```

You'll see the following prompt:

```
OBJECT FILE (CR OR D:NAME):
```

Now remove the assembler/editor disk from the disk drive, replace it with your own program disk, and type the file name:

```
ADDNRS.ASM
```

The assembler will assign that name to the object file it will soon be creating.

Next, you're asked whether you want a hardcopy (printout) of the assembly code listing. If you do, press Return. If you don't, type N.

Your assembler/editor now asks if you want another kind of file called a cross-reference file (so you can refer in other programs to any labels, constants, or other identifiers you've created in this one). You can't create an object code file and a cross-reference file during the same pass through the Commodore 64 assembler, and you'll have no need for such a file at this point in your study of assembly language. So press Return.

Finally, you're asked to give the name of the source code program you want assembled. In response to this prompt, type:

```
ADDNRS.SRC
```

The C-64 assembler then assembles the program and provides you with a listing—either on the screen or on paper, depending on what you requested—that looks like the one in listing 4-8.

Listing 4-8 Assembled listing of ADDNRS.SRC program

```
ADDNRS.SRC......PAGE 0001
LINE# LOC   CODE         LINE

00001  0000    ;
00002  0000               ;ADDNRS.SRC
00003  0000               ;
00004  0000                      *=$1300
00005  1300               ;
00006  1300  D8           ADDNRS CLD
00007  1301  18                  CLC
00008  1302  A9 02               LDA #2
00009  1304  69 02               ADC #2
00010  1306  8D 00 0C            STA $0C00
00011  1309  60                  RTS
00012  130A                      .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE

 ADDNRS   1300

END OF ASSEMBLY
```

Finding Your Errors

If you've made any typing errors, this is where you may find out about them. If the assembler detects an error while it's assembling a program, it flags the mistake and displays an error message. It may not be able to spot every error you make, but when it does catch one, it will print an error message on the screen or the printout. If you don't understand what the message means, consult the C-64 assembler/editor user's manual.

As the assembler assembles your program, it automatically saves an object code listing on the disk in the disk drive, using the file name ADDNRS.ASM. If the assembler finds any errors in your program while the program is being assembled and saved, it still saves the assembled program—errors and all.

If there are any errors in your program, you'll have to reload the EDITOR64 program so you can go back to the original source code listing. Then you have to take the following steps, in the correct order: Erase the incorrectly written program from the disk (hope you're using the wedge), reload the assembler, assemble the source code again, and see whether you've made any more errors. If there are more mistakes, guess what? You'll have to go through each of the preceding steps again (whew!), and keep going through them until all of the errors that the assembler has spotted are removed from the program.

When you've assembled the ADDNRS.SRC program, and the assembler has saved the assembled version of the program on a disk, you're ready to convert the assembled program into true machine language using the C-64 loader utility. But we'll save that step for the next chapter.

TSDS Assembler

The TSDS (Total Software Development System) assembler combines some of the features in the Merlin 128 assembler with many of the features in the Commodore 64 Macro Assembler System. The TSDS assembler is not a menu-driven system; instead, it works much like the BASIC editor built into the C-128. When the TSDS system is up and running, you don't have to choose operating modes from a menu; simply type in a command line, and the assembler executes the command immediately. And, with the TSDS system, you can mix DOS commands and assembler commands quite freely; when the TSDS assembler is running, you can use DOS commands such as DLOAD, DSAVE, BLOAD, BSAVE, SCRATCH, and NEW, and they will be executed just as they would be if you were using the C-128's built-in BASIC 7.0 interpreter.

ADDNRS.SRC Program (TSDS Version)

Listing 4-9 shows how the ADDNRS program looks when it's typed on a TSDS assembler.

Listing 4-9 ADDNRS.SRC program, TSDS version

```
10 ;
20 ; ADDNRS.SRC
30 ;
40  *=$1300
50 ;
60 ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
100  STA $0C00
110  RTS
```

Line Numbers

The NoSync TSDS assembler, like the Merlin 128 assembler, loads automatically when booted. It allows you to type assembly language programs in much the same way that you type BASIC programs. The TSDS assembler does not generate relative line numbers, as the Merlin 128 assembler does. Instead, it allows you to write your own line numbers—starting with any number, ending with any higher number, and progressing in any increments you desire.

Another important feature of the TSDS assembler is that it uses a full-screen editor. When you create a program with TSDS, you can use the C-128's arrow keys to move the cursor anywhere on the screen—and, by placing the cursor over lines that have already been typed, you can make corrections without retyping the lines.

If you have trouble being neat and tidy when you put spaces in programs, you may be happy to hear that TSDS is not finicky about spacing. In fact, you can put labels and mnemonics underneath each other in a program, without any indentations, although it may make your programs difficult to read. TSDS will sort out the mess when it assembles your program.

TSDS, like all assemblers, uses its own set of pseudo op codes, and they are different from the op codes recognized by both Merlin and the C-64 Macro Assembler. TSDS recognizes an asterisk followed by an equal sign (*=) as a directive to set its program counter, and uses the .END directive to end programs. The .BYTE directive used by the C-64 Macro Assember (which is the same as the DBF or DB directive used by Merlin) becomes .BYT in the TSDS assembler's set of pseudo op commands. You can store 16-bit words in memory with the .WOR ("word") and .DBY ("double byte") directives.

When you type a program using TSDS, you can delete blocks of lines with the command DELETE, just as you can in BASIC 7.0 programs. But there's no easy way to move blocks of code from one part of a program to another, as there is when you use the Merlin assembler.

Assembling the Program

To assemble a program that has been typed using TSDS, use the ASM command. If you use the ASM command by itself, TSDS assembles the program but won't save it on a disk. To instruct TSDS to assemble the program and save it, you must follow the ASM command with TO and a file name, as follows:

```
ASM TO "ADDNRS.OBJ"
```

If you typed an assembly language program using TSDS and then typed the preceding command line, the program would be assembled and saved to disk under the file name ADDNRS.OBJ.

Saving the Program

When you have typed a source code listing using TSDS, you can save it as a sequential file by typing the command PUT, followed by a file name. The file name can be enclosed in quotation marks if desired, but quotation marks are not necessary. Thus, a source code listing can be saved as a sequential file by typing a command such as:

```
PUT ADDNRS.SRC
```

or a command such as:

```
PUT "ADDNRS.SRC"
```

The form you choose doesn't matter to the TSDS assembler.

After a source code program is saved as a sequential file, it can be loaded using the command GET, followed by the name of the desired file. Again, the file name may be enclosed in quotation marks if desired, but quotation marks are not necessary.

To save a source code listing as a text file, so that it can be edited using a word processor or transmitted as text, use the DOS command DSAVE, followed by the file name. To load a source code program that has been saved as a text file, use the DOS command DLOAD. The commands BSAVE and BLOAD can be used to save and load binary files created using the TSDS assembler. File names used with the commands DLOAD, DSAVE, BLOAD, and BSAVE must be enclosed in quotation marks.

The TSDS assembler works in both 40-column mode and 80-column mode on a Commodore 128. To list a TSDS program on the C-128 screen, type the command LIST. To print a hardcopy of a source code listing, all you have to do is precede the word LIST with an asterisk, as follows:

```
*LIST
```

Some useful extras come with the TSDS system, including a sprite editor, a character and shape editor, and some sample source code programs.

C-128 Monitor

We wrote a short program using the Commodore 128 monitor in chapter 1, so there's no need to repeat the information presented in that chapter. But there are a few additional comments that could be said about the C-128 monitor, so I'll say them before we move to chapter 5.

As you may recall from chapter 1, you can activate the C-128 monitor by holding down the shift key while you start your computer, by pressing the reset button, or by typing the command MONITOR while BASIC is running. The screen that is displayed when the monitor is activated is illustrated in figure 4-2.

Figure 4-2
C-128 monitor's opening display

```
    PC    SR AC XR YR SP
; FB000   00 00 00 00 F8
```

The top line of the display in figure 4-2 is a list of the 8502 processor's six internal registers: the program counter, the processor status register, the accumulator, the X register, the Y register, and the stack pointer. The functions of all of these registers are described in chapter 3. The second line of the display shows, from left to right: the memory bank currently being accessed (in this case, bank F); the current contents of the 8502 program counter (in this case, $B000); and the current states of the 8502's other five registers.

Assembling the Program

As soon as the C-128 monitor is called and its opening display appears on the sceen, it is ready to assemble a program. To instruct the monitor to start assembling a program, type A, type the bank number in which the program is to be assembled, and then type the program's starting address. Then the source code that is to be assembled can be entered. For example, the following line could be typed to start assembling the ADDNRS.S program in memory bank 0, starting at memory address $1300:

```
01300 LDA #2
```

When a source code listing is typed using the C-128 monitor, the monitor assembles each line as soon as the Return key is pressed, and stores the line in memory. As each line of a program is assembled into machine language and stored in memory, the monitor generates an onscreen listing of the code which it has assembled. Listing 4-10 shows what kind of listing is generated if the ADDNRS.SRC program is typed and assembled using the C-128 monitor. The first column shows memory banks and addresses, the second column is the object code (which is generated by the C-128 monitor), and the third column is the source code (what you type in).

Listing 4-10 ADDNRS.SRC program, C-128 monitor version

```
01300      D8              CLD
01301      18              CLC
01302      A9 02           LDA #$02
01304      69 02           ADC #$02
01306      8D 00 0C        STA $0C00
01309      60              RTS
0130A                      .END
```

Saving the Program

The C-128 monitor cannot save or load the source code listings of assembly language programs. However, after a program is written and assembled using the monitor, the monitor can save the object code version of the program. After an object code program is saved on a disk, the monitor can load the machine code program into memory and then disassemble it back into assembly language.

When you have written a source code program and the C-128 assembler has assembled it, you can save the program's object code by typing the monitor command S. The command must be followed by a file name in quotation marks, a device number (usually the number 8 if a disk drive is being used), the starting address of the program, and the ending address of the program plus 1. The following command line could be used to save the object code version of the ADDNRS program:

```
S "ADDNRS.OBJ",8,1300,130B
```

After you have saved an object code program on a disk—either with the monitor or with an assembler or some other utility—you can load it into memory using the monitor command L. This command must be followed by the file name of the program and the number of the device from which it is being loaded. The following command line could be used to load the object code version of the ADDNRS program:

```
L "ADDNRS.OBJ",8
```

Limitations

The C-128 monitor is a handy programming utility, but its built-in miniassembler has many limitations. For example, as we've mentioned, it does not recognize labels and cannot be used to load or save source code programs. That's why it's called a miniassembler.

Despite its limitations, however, the C-128 monitor can still perform many useful functions. For example, it can be used to load and execute machine language programs, as we'll see in the next chapter. Before it can do that, though, the program to load and execute must be stored on a disk. So, if you haven't yet assembled the ADDNRS program and stored it on a disk, this might be a good time to do so. Then you'll have it handy for the next chapter.

5

Running an Assembly Language Program

From a BASIC program, or on its own

After an assembly language program has been assembled into machine language, there are three ways you can execute it:

- from DOS
- from BASIC
- from the C-128 machine language monitor

To execute a machine language program from DOS, you can load the program into memory using the command BLOAD, and then execute it using either a SYS or USR(X) call. To load and execute a machine language program from BASIC, you can perform the same two procedures, but from within a BASIC program. Or you can execute a machine language program from the C-128 monitor by using the monitor's G command.

In this chapter, we'll be covering all three of these methods of executing a machine language program. We'll start with the the third method: running a machine language program using the C-128 monitor.

Running a Program from the C-128 Monitor

The machine language monitor that's built into the Commodore 128 is extremely powerful, very versatile, and not difficult to use. As you may recall from chapter 4, to invoke the monitor just type the command MONITOR and press Return. The monitor's opening display then appears on your screen, as follows:

```
    PC   SR AC XR YR SP
; FB000 00 00 00 00 F8
```

and you'll be ready to go.

If you followed the instructions at the end of chapter 4, you now have the ADDNRS.OBJ program stored on a disk. So, after you have activated the C-128 monitor, you can use the monitor command L to load the ADDNRS.OBJ code into memory. Just put your disk into the disk drive and, on the line that follows your monitor's opening display, type:

```
L "ADDNRS.OBJ",8
```

To see if the program is loaded, you can now type:

```
D 1300 130A
```

which means "disassemble memory addresses 1300 through 130A." In assembly language jargon, to dissassemble means to convert a segment of machine code back into source code. So, when you type the preceding line, the Commodore 128 should respond with a disassembled (source code) listing of the ADDNRS program.

If you now have your monitor up and running, you can easily run the ADDNRS program, too. On the next line on the screen, type:

```
G 1300
```

and press Return. The computer will respond with another READY prompt—which means that something has definitely happened because you've now exited the monitor and are back in BASIC!

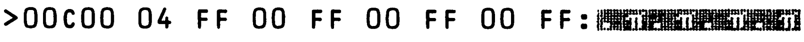
Why is that? It's because the ADDNRS program ends with an RTS instruction. When the C-128 monitor encounters an RTS instruction without any address to return to, it returns control to BASIC. There is, by the way, an assembly language instruction that can prevent the Commodore monitor from returning to BASIC after it finishes running a program using the G command. The instruction that can keep this from happening is the mnemonic BRK (for "break"). Programmers often use the BRK mnemonic when they're debugging programs. By putting a BRK instruction at the end of an assembly language routine, you can debug your program without worrying about the assembler jumping back into BASIC every time it comes to the end of the routine. When you've finished debugging a program, though, remove any BRK instructions that have been used for debugging because they can crash the program when the program is run outside a monitor environment.

But that's not our problem now. We want to get our assembler back into monitor mode. Type MONITOR—the same BASIC 7.0 command that you used to activate the monitor in the first place—and the assembler returns to monitor mode.

When your monitor's opening screen display reappears, you can check to see whether the monitor has executed the ADDNRS program successfully. Type:

```
M 0C00 0C07
```

Your monitor should respond with a line that looks like figure 5-1.



```
>00C00 04 FF 00 FF 00 FF 00 FF:
```

Figure 5-1
C-128 monitor's M display

Figure 5-1 is a listing of the contents of memory location $0C00 and the next seven addresses in the computer's memory. The symbols that follow the hexadecimal numbers are the ASCII equivalents of the hex numbers. They don't mean much to us now, but they can come in handy when the M command is used to view ASCII data. In this case, though, all we want to know is whether the ADDNRS.OBJ program worked. And, because the first number after the address in figure 5-1 is 04, we can see that we've succeeded; the program has added 2 and 2, and has stored the sum in memory address $0C00!

Running a Program from DOS

We'll use a new program to illustrate how programs written in assembly language can be run using DOS and BASIC commands. The next program is called HITEST.S, and when you assemble and execute it you'll see why. Then we'll take a look at how it works.

Listing 5-1 shows what the HITEST.S program looks like when it is typed using the Merlin 128 assembler. If you own some other kind of assembler, it shouldn't be too difficult to figure out what kinds of modifications are needed to make it compatible with your system.

Listing 5-1
HITEST.S program

```
*
* HITEST.S
*
          ORG    $1300
*
          LDA    #72
          JSR    $FFD2
          LDA    #73
          JSR    $FFD2
          RTS
```

How It Works

HITEST.S is a short program with a big secret; it's only five lines long, yet it can print a message on your C-128's screen—a job that would ordinarily require many more lines of code. The program can do such a big job with so few instructions because it uses the C-128 kernel: a collection of useful machine language subroutines that can be incorporated into user-written programs quickly and easily.

The C-128 kernel resides in a segment of ROM that extends from memory address $E000 through memory address $FFFF. After you know how to use it, you can perform many kinds of input/output operations by simply calling a prewritten subroutine, rather than writing the subroutine yourself. The kernel contains routines that can read a character typed in at the keyboard, print a character on the screen, switch to C-64 mode or 80-column mode, and perform many kinds of file operations.

The C-128 kernel is guaranteed to be compatible with future versions of the computer, so if you write programs using the kernel, they won't become obsolete when models change. In fact, the Commodore 64 also has a kernel, and the C-128 kernel is downwardly compatible with the kernel built into the C-64. The C-128 kernel contains more subroutines than the C-64 kernel, but the C-64 kernel is a totally compatible subset of the kernel built into the C-128.

Most of the subroutines built into the C-128 kernel are very easy to access. To use a kernel subroutine, all you usually have to do is place certain values in one or more 8502 registers. Then you can use the assembly language instruction JSR to jump to the address of the desired subroutine. Another important feature of the kernel is that it can save you memory because it eliminates the necessity of duplicating routines that are already built into ROM.

The subroutine that is called by the HITEST.S program has a call address of $FFD2 and is titled BSOUT. The same subroutine was included in the Commodore 64 kernel, but in its C-64 version it was called CHROUT. To use the BSOUT subroutine, place an ASCII value in the 8502 accumulator and then do a JSR to memory address $FFD2. The character that equates to the ASCII code in the accumulator will then be printed on the screen.

A complete list of the C-128 kernel subroutines can be found in the *Commodore 128 Programmer's Reference Guide* and in other C-128 reference manuals. A number of kernel routines are used in other programs in this book.

Executing the HITEST.S Program

After a program is assembled and saved on a disk, it is easy to execute the program using a DOS command line. For example, if you have assembled the HITEST.S program into a machine language program titled HITEST.O, you can execute the program by typing the command:

```
BLOAD "HITEST.O"
```

followed by the command:

```
SYS 4864
```

If you want your C-128 to do the job of converting the address of the HITEST.O program into a decimal number, and if you want to call the program using a one-line command, type:

```
BLOAD "HITEST.O":SYS DEC("1300")
```

and the result is the same.

Running a Program from BASIC

It is a little more difficult to execute a machine language program from BASIC than it is to run it using a DOS command line. But it's worth the extra trouble because you only have to do the work once.

After you set up a BASIC program that calls a machine language program, the machine language program loads and executes automatically each time the BASIC program is run, and you never again have to worry about the machine language program's address, or about writing a complicated DOS command line.

One problem that often arises when a machine language program is called from BASIC is that the BLOAD command sometimes sets up an endless loop that loads the machine language program into memory over and over again. Here is why that happens.

When a BLOAD instruction is encountered in a BASIC program, a search is made for the requested machine language program. If the program is found, it is loaded into memory. Then, control of the computer returns to BASIC. If a BASIC program is found in memory, the C-128 starts running it from the beginning—even if it is the BASIC program that was already running when the search for the machine language program began.

This is where the endless loop begins. The BASIC program starts running, and when it gets to the line that contains the BLOAD command, the machine language program is loaded into memory again. And again, and again, and again.

Chasing the Endless Loop

To prevent an infinite loop from being set up, C-128 programmers often use an ingenious trick: They write a short BASIC routine that looks like the one shown in listing 5-2.

Listing 5-2 Calling a machine language program from BASIC

```
10 IF X=0 THEN X=1:BLOAD "HITEST.O"
20 SYS DEC("1300")
```

When a BASIC routine like the one in listing 5-2 is executed, it first checks the value of a variable labeled X. Ordinarily, when a BASIC program starts running, the value of any variable that has not been defined is 0. So the IF...THEN statement in line 10 of listing 5-2 is interpreted as true, and all of the operations that follow the instruction THEN are carried out. This means that the value of X is changed to 1 and that the HITEST.O program is loaded into memory.

When HITEST.O is loaded, control returns to BASIC, and the program in listing 5-2 starts running again, beginning at line 10. But this time, the value of X is 1, not 0—so HITEST.O is not loaded this time, and an infinite loop does not begin. Instead, the program moves to line 20, and the SYS command in that line executes the HITEST.O program, beginning at its starting address: $1300, or 4864 in decimal notation.

Using Parameters with the SYS Call

One new feature of the Commodore 128 is that it allows you to use up to four parameters, in addition to a starting address, with the SYS call. When optional parameters follow a SYS command, their values are loaded into four of the 8502's internal registers as soon as the desired machine language program is called. If optional parameters are used with the SYS call, they must be separated by commas. Following is the syntax of the C-128 SYS command:

SYS *address{,accumulator,X register,Y register,status register}*

Optional values, which are enclosed in braces, can be omitted if desired. For example, the X register and the status register can be omitted by using the following syntax:

```
SYS 4864,42,,9
```

COLORME2.S, the program in listing 5-3, is designed to be called by a SYS command followed by three parameters: a value to be loaded into the accumulator, a value to be loaded into the X register, and a value to be loaded into the Y register.

Listing 5-3
COLORME2.S
program

```
*
* COLORME2.S
*
         ORG    $1300
*
         STA    $D020
         STX    $D021
         STY    $F1
         RTS
```

COLORME2.S is a revised version of COLORME64.S, the first program presented in this book (at the beginning of chapter 1). COLORME64.S, as you may recall, changed the colors of the C-128 screen to imitate the screen colors of the C-64. COLORME2.S performs a similar operation, but because you can use parameters with the SYS command that calls the program, COLORME2.S can change the screen colors to any colors desired.

When you have typed, assembled, and saved the COLORME2.S program, you can execute it with this type of system call:

```
SYS DEC("1300"),14,6,14
```

This call changes your C-128's screen colors to those of a C-64, just as the COLORME64.S program did. But, by using different parameters after the SYS command, you can select different colors.

RREG Instruction

BASIC 7.0 also has a handy instruction, RREG, that can can read the 8502 chip's A, X, Y, and S registers. When a machine language program is executed from BASIC, the RREG instruction can tell you what values these four registers hold at the conclusion of the program.

Listing 5-4 is a BASIC program that uses the RREG instruction to read the values of the 8502's A, X, Y, and S registers.

Listing 5-4 Using the RREG instruction

```
10 REM *** READ 8502 REGISTERS ***
20 PRINT "{CLR}"
30 PRINT "THIS PROGRAM WILL READ"
40 PRINT "YOUR 8502 REGISTERS"
50 PRINT
60 RREG A,X,Y,S
70 PRINT "ACCUMULATOR:";A
80 PRINT "X REGISTER:";X
90 PRINT "Y REGISTER:";Y
100 PRINT "STATUS REGISTER:";S
```

USR(X) Function

SYS is not the only reserved BASIC 7.0 word that can be used to execute a machine language program. BASIC also has a function, USR(X), that can call a machine language program or a machine language routine.

Before the USR(X) function can call a machine language program, the program must be loaded into memory. Then the starting address of the machine language program must be stored, low byte first, in memory addresses $1219 and $121A (4633 and 4634 in decimal notation). Then the USR(X) function can be invoked using one of the following formats:

Y=USR(*X*)

where *Y* and *X* are two separate variables, or:

Y=USR(*n*)

where *Y* is a variable and *n* is a numeric value.

If you use the format:

Y=USR(*X*)

the *X* variable can be either a dummy value—which doesn't have to equate to anything and doesn't do anything—or a value that will be passed to the machine language program.

If you use the format:

Y=USR(*n*)

the *n* value can be passed to the machine language program, and the program can then perform any desired function using that value. When the machine language program ends and control returns to BASIC, the machine language program can pass the result of its calculations back to BASIC, as the value of the *Y* variable.

When the USR(X) function is executed, the value inside the parentheses is stored in an area of memory called floating-point accumulator 1, or FAC1. Actually, there are two floating-point accumulators in the C-128; one is called FAC1 and the other is called FAC2. When BASIC performs an arithmetical problem, it stores the values that it is working with in FAC1 and FAC2. As their names imply, FAC1 and FAC2 store numbers in floating-point fashion, in a format similar to the one that was described in chapter 2.

At this point, the use of the USR(X) function gets a little complicated. Because floating-point numbers are not customarily used in user-written programs, it is often necessary to convert the number stored in FAC1 into a binary number before a desired USR(X) function can take place. Fortunately, the C-128 has several built-in subroutines that can convert floating-point numbers to binary integers, and vice versa. One such subroutine, called GETADR, can be called by doing a JSR to memory address $AF0C.

When GETADR is called, it expects a floating-point value to be stored in FAC1. GETADR converts that value into a binary integer, and stores the binary integer, low byte first, in memory addresses $16 and $17 (or 22 and 23 in decimal notation). The machine language program can then perform any desired operation on the value stored in $16 and $17, using ordinary binary arithmetic.

When the machine language program has finished its calculations, it can convert the result of its operations into a floating-point number before returning control to BASIC, thus ensuring that the result of the calculation is passed back to BASIC in a form that BASIC "understands."

An easy way to convert a binary value into a floating-point value is to use another subroutine, called GIVAYF, which can be called by doing a JSR to memory address $AF03. When GIVAYF is called, it expects a 16-bit binary value to be stored in the accumulator and the Y register, with the low byte in the Y register and the high byte in the accumulator. GIVAYF converts these two values into a floating-point number, and stores the number in FAC1. So, when the USR(X) function is used to pass a value to a machine language program and then get a value back, both the number passed to the machine language program and the number returned to BASIC can be found in FAC1.

Listing 5-5, a program titled USRDEMO.SRC, shows how the USR(X) function can pass a value to a machine language program, and how the machine language program can perform an operation on the value and pass the result of the operation back to BASIC.

USRDEMO.BAS, a BASIC program that calls USRDEMO.SRC, appears in listing 5-6.

Listing 5-5
USRDEMO.SRC program

```
1000 ;
1010 ; USRDEMO.SRC
1020 ;
1030  *=$0C00
1040 ;
1050 START JSR $AF0C ;PUT FAC1 IN $16 AND
     $17
1060  CLC ;PREPARE FOR ADDITION
1070  LDA $16 ;GET LOW BYTE OF X
1080  ADC #5 ;ADD 5
1090  TAY ;PLACE LOW BYTE OF SUM IN Y
      REGISTER
1100  LDA $17 ;GET HIGH BYTE OF X
1110  ADC #0 ;ADD CARRY, IF ANY
1120  JMP $AF03 ;PLACE Y AND A IN FAC1 AND
      RETURN
1130 ;
```

Listing 5-6
USRDEMO.BAS program

```
10 REM *** USRDEMO.BAS ***
15 IF A=0 THEN A=1:BLOAD "USRDEMO.OBJ"
20 INPUT "TYPE A NUMBER";X
30 POKE 4633,DEC("00"):POKE 4634,DEC("0C")
40 A=USR(X)
50 PRINT "THE SUM OF";X;"AND 5 IS";A
```

This chapter is much shorter than chapter 4, but that isn't surprising because it's much easier to run an assembly language program than it is to write one. And that's okay because a short chapter means we can move on to the next chapter that much sooner. This brings us to chapter 6, which is all about the addressing modes used in 6502/8502 assembly language.

6

Addressing the Commodore 128

Addressing modes of the 8502

In chapter 1, we saw that there is a one-to-one correlation between assembly language and machine language. For every mnemonic in an assembly language program, there's a numeric machine language instruction that means the same thing. We also saw in chapter 1 that although that's the truth, it isn't quite the whole truth. Actually, most instructions used in 8502 assembly language have more than one equivalent instruction in machine language. For example, the mnemonic ADC in an assembly language program can be converted into eight different numeric instructions when it is assembled into machine language. To understand why this is true, you need to know how addressing modes are used in 6502/8502 assembly language.

In the world of assembly language programming, an addressing mode is a tool for locating and using information stored in a computer's memory. The Commodore 128's 8502 chip can access memory locations in the computer in thirteen ways. So the 8502 processor in the Commodore 128 has thirteen addressing modes.

In this chapter, we'll examine all thirteen addressing modes, and observe how they are used in 6502/8502 assembly language. First, though, let's take a look at the eight ways that one mnemonic—ADC—can be converted into machine language. Table 6-1 shows the eight addressing modes that can be used with the ADC mnemonic.

Table 6-1
Addressing Modes of ADC Mnemonic

Column 1	Column 2	Column 3	Column 4
	Sample Assembly		
	Language	Machine Code	Number of
Addressing Mode	Statement	Equivalent	Bytes
Immediate	ADC #$03	69 03	2
Zero Page	ADC $03	65 03	2
Zero Page,X	ADC $03,X	75 03	2
Absolute	ADC $0300	6D 00 03	3
Absolute Indexed,X	ADC $0300,X	7D 00 03	3
Absolute Indexed,Y	ADC $0300,Y	79 00 03	3
Indexed Indirect	ADC ($03,X)	61 03	2
Indirect Indexed	ADC ($03),Y	71 03	2

Later in this chapter, we'll examine all of the addressing modes listed in table 6-1, and see how they work in assembly language programs. First, though, let's compare the assembly language statements and machine language statements listed in table 6-1.

In column 2 of table 6-1, the assembly language column, all eight statements have the same mnemonic, but each has a different operand. But in column 3, the machine language column, the statements have a different structure. In the machine language column, there are eight different op codes but only two operands: the one-byte operand 03, and the two-byte operand 00 03.

This arrangment illustrates an important difference between assembly language and machine language: a difference that we first observed in chapter 1. In 6502/8502 assembly language, addressing modes are distinguished by differences in their op codes. But in

6502/8502 assembly language, the thirteen available addressing modes can be identified by differences in their operands.

Table 6-2 shows the thirteen addressing modes that are recognized by the 8502 chip.

Table 6-2
Addressing Modes of 8502

Addressing Mode	Format
Implicit (Implied)	RTS
Accumulator	ASL A (or ASL)
Immediate	LDA #2
Absolute	LDA $0C00
Zero Page	STA $FA
Relative	BCC LABEL
Absolute Indexed,X	LDA $0C00,X
Absolute Indexed,Y	LDA $0C00,Y
Zero Page,X	LDA $FA,X
Zero Page,Y	STX $FA,Y
Indexed Indirect	LDA ($FA,X)
Indirect Indexed	LDA ($FA),Y
Indirect	JMP

ADDNRS.SRC Revisited

Listing 6-1 uses ADDNRS.SRC, an 8-bit addition routine that you may recall from chapter 1, to show how some of the 8502's addressing modes work. The program was typed using a TSDS assembler, as were the rest of the programming examples in this chapter. (If you have a Merlin or Commodore assembler, or some other assembler/editor system, you can alter the program to meet your system's demands without too many problems by now because the important differences in the formats used by these assemblers were described in chapters 4 and 5).

Listing 6-1
ADDNRS.SRC program

```
1000 ;
1010 ; ADDNRS.SRC
1020 ; AN 8-BIT ADDITION PROGRAM
1030 ;
1040 *=$1300
1050 ;
1060 ADDNRS CLD ;IMPLIED ADDRESS
1070  CLC ;IMPLIED ADDRESS
1080  LDA #02 ;IMMEDIATE ADDRESS
1090  ADC #02 ;IMMEDIATE ADDRESS
1100  STA $FA ;ZERO PAGE ADDRESS
1110  RTS ;IMPLIED ADDRESS
1120  .END
```

In listing 6-1, the three addressing modes used in the ADDNRS.S program are identified in the comments column. Let's look now at each of these three addressing modes.

Implied (or Implicit) Addressing

In implied (implicit) addressing mode, the operand is not written but understood. When you use implied addressing, you only have to type the three-letter assembly language instruction; implied addressing does not require (in fact does not allow) the use of an expressed operand.

Op code mnemonics that can be used in implied addressing mode are BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

Immediate Addressing

When immediate addressing is used in an assembly language instruction, the operand that follows the op code mnemonic is a literal number—not the address of a memory location. Therefore, in a statement that uses immediate addressing, a # symbol (the symbol for a literal number) always appears before the operand.

When an immediate address is used in an assembly language statement, the assembler does not have to PEEK into a memory location to find a value. Instead, the value itself is placed directly into the accumulator. Then, the operation the statement calls for can be immediately performed. Instructions that can be used in immediate address mode are ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, and SBC.

Zero Page Addressing

It is very easy to distinguish between a statement that uses immediate addressing and one that uses zero page addressing. In a statement that uses zero page addressing, the operand always consists of one byte—a number ranging from $00 to $FF. And that number always equates to an address in a block of RAM called page zero.

Page zero is a 256-byte block of RAM that extends from memory address $00 through memory address $FF. Every memory location on page zero has a 1-byte address, and thus can be addressed using a 1-byte operand. Another noteworthy fact about page zero is that some addressing—as we shall see later in this chapter—actually requires zero page operands. Because the 256 memory addresses on page zero are so valuable, page zero is the high-rent district in your Commodore's RAM; it's such a desirable piece of real estate, in fact, that the people who designed the computer took most of it for themselves. Most of page zero is used by the computer's operating system and other essential routines, and not much space is left for user-written programs.

Because space on page zero is so useful and so scarce, designers of 6502-based computers have been trying for years to increase the amount of storage space available on page zero. And, in designing the

Commodore 128, they finally succeeded. As we will see in chapter 10, which is devoted to memory management and the memory layout of the C-128, designers of C-128 programs can relocate page zero to any other page in memory. For now, the most important fact to remember about page zero is that it's an addressing mode that uses a memory address on page zero as a 1-byte operand.

Instructions that can be used with zero page addressing are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

New Addressing Modes

Now we'll describe the five 6502/8502 addressing modes that we haven't covered yet.

Accumulator Addressing

Accumulator addressing mode is used to perform an operation on a value stored in the 6502/8502 processor's accumulator. When accumulator addressing mode is used, some assemblers require the letter A as an operand; other assemblers, including both the Merlin 128 and the TSDS assembler, do not require the A operand. One example of a statement that uses accumulator addressing mode is ASL, or ASL A. This statement rotates each bit in the accumulator one position to the left, with the leftmost bit (bit 7) dropping into the carry bit of the processor status (P) register. Other instructions that can be used in accumulator addressing mode are LSR, ROL, and ROR.

Absolute Addressing

Absolute addressing is very similar to zero page addressing. In a statement that uses absolute addressing, the operand is a memory location, not a literal number. The operation called for in an absolute address statement is always performed on the value stored in the specified memory location, not on the operand itself.

The difference between an absolute address and a zero page address is that an absolute address statement doesn't have to be on page zero; it can be anywhere in free RAM. So an absolute address statement requires a 2-byte operand—not a 1-byte operand, which is all that a zero page address requires.

Listing 6-2 shows what the ADDNRS.S program looks like when we use absolute addressing instead of zero page addressing.

Listing 6-2
ADDNRS2.SRC
program

```
1000 ;
1010 ; ADDNRS2.SRC
1020 ; AN 8-BIT ADDITION PROGRAM
1030 ;
1040 *=$1300
1050 ;
1060 ADDNRS CLD ;IMPLIED ADDRESS
1070  CLC ;IMPLIED ADDRESS
1080  LDA #02 ;IMMEDIATE ADDRESS
1090  ADC #02 ;IMMEDIATE ADDRESS
1100  STA $0C00 ;ABSOLUTE ADDRESS
1110  RTS ;IMPLIED ADDRESS
1120  .END
```

The only new feature in listing 6-2 appears in line 1100. The operand in that line is now a 2-byte operand, and that change makes the program one byte longer. But now the address in line 1100 doesn't have to be on page zero; it can be the address of any free byte in RAM.

Mnemonics that can be used in absolute addressing mode are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

Relative Addressing

Relative addressing mode is used for a technique called conditional branching—a method for instructing a program to jump to a given routine under specific conditions. There are eight conditional branching instructions, or relative address mnemonics, in 6502/8502 assembly language. All eight begin with B, which stands for "branch to." Examples of conditional branching instructions that use relative addressing are: BCC ("branch to a specified address if the carry flag is clear"), BCS ("branch to a specified address if the carry flag is set"), BEQ ("branch to a specified address if the result of an operation is equal to zero"), and BNE ("branch to a specified address if the result of an operation is not equal to zero"). All eight conditional branching instructions are described in chapter 7, which is devoted to the topic of looping and branching.

The eight conditional branching instructions are often used with three other instructions called comparison instructions. Typically, a comparison instruction is used to compare two values with each other, and the conditional branch instruction is then used to determine what to do if the comparison turns out in a certain way.

The three comparison instructions are: CMP ("compare the number in the accumulator with..."), CPX ("compare the value in the X register with..."), and CPY ("compare the value in the Y register with..."). Conditional branching instructions can also follow arithmetical operations, logical operations, and various kinds of testing of bits and bytes.

Usually, a branch instruction causes a program to branch to a specified address if certain conditions are met or not met. A branch might be made, for example, if one number is larger than another, if the two numbers are equal, or if an operation results in a positive, negative, or zero value.

Listing 6-3, titled ADD8BIT.SRC, is an example of an assembly language routine that uses conditional branching.

Listing 6-3
ADD8BIT.SRC
program

```
1000 ;
1010 ; ADD8BIT.SRC
1020 ;
1030  *=$1300
1040 ;
1050 ADD8BIT LDA #0
1060  STA $0C04 ;CLEAR ERROR FLAG
1070 ;
1080  CLD ;CLEAR DECIMAL FLAG
1090  CLC ;CLEAR CARRY FLAG
1100 ;
1110  LDA $0C00
1120  ADC $0C01 ;ADD $0C00 AND $0C01
1130  BCS ERROR ;WAS THERE A CARRY?
1140  STA $0C03 ;NO, STORE RESULT
1150  JMP FINI ;AND END PROGRAM
1160 ;
1170 ERROR LDA #1 ;YES ...
1180  STA $0C04 ;SO SET CARRY FLAG
1190 ;
1200 FINI RTS ;END OF PROGRAM
```

ADD8BIT.SRC is an 8-bit addition program with a simple error-checking utility. It adds two 8-bit values using absolute addressing. If this calculation results in a 16-bit value (a number larger than 255), there is an overflow error in addition, and the carry bit of the processor status register is set.

If the carry bit is not set, the sum of the values in $0C00 and $0C01 is stored in $0C03. If the carry bit is set, however, this condition is detected in line 1130, and the program branches to the line labeled ERROR (line 1170). If an error is detected, the values in $0C00 and $0C01 are not added. Instead, a flag (the number 1) is loaded into memory address $0C04, and the routine ends.

Absolute Indexed Addressing

An indexed address, like a relative address, is calculated by using an offset. But in an indexed address, the offset is determined by the current contents of the 6502/8502's X register or Y register.

A statement containing an indexed address can be written using either of these formats:

```
LDA $0C00,X
LDA $0C00,Y
```

When indexed addressing is used in an assembly language statement, the contents of either the X register or the Y register (depending upon which index register is being used) are added to the address given in the instruction to determine the final address.

Listing 6-4 is an example of a routine that makes use of indexed addressing. The routine is designed to move byte by byte through a string of ASCII characters, storing the string in a text buffer. When the string is stored in the buffer, the routine ends. The text to be moved is labeled TEXT, and the buffer to be filled with text is labeled TXTBUF. The starting address of TXTBUF, plus the ASCII number for a carriage return, are defined in a symbol table that precedes the program.

Listing 6-4
DATMOV.SRC
program

```
1000 ;
1010 ;DATMOV.SRC
1020 ;
1030  TXTBUF = $0C00
1040  EOL = $0D
1050 ;
1060 *=$1300
1070 ;
1080  JMP DATMOV
1090 ;
1100 TEXT .BYT $54,$41,$4B,$45,$20,$4D,$45,
     $20
1110  .BYT $54,$4F,$20,$59,$4F,$55,$52,$20
1120  .BYT $4C,$45,$41,$44,$45,$52,$21,$0D
1130 ;
1140 DATMOV
1150 ;
1160  LDX #0
1170 LOOP LDA TEXT,X
1180  STA TXTBUF,X
1190  CMP #EOL
1200  BEQ FINI
1210  INX
1220  JMP LOOP
1230 FINI RTS
1240  .END
```

When the program begins, we know that the string ends with an end-of-line (EOL) character, the ASCII character $0D, which is generated by the Return key on the C-128 keyboard.

As the program proceeds through the string, it tests each character to see whether it is a carriage return. If the character is not a carriage return, the program moves to the next character. If the character is a carriage return, it means there are no more characters in the string, and the routine ends.

Zero Page,X Addressing

Zero page,X addressing is used like absolute indexed,X addressing. However, the address used in zero page,X addressing mode must be located on page zero (logically enough). Instructions that can be used in zero page,X addressing mode are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, and STY.

Zero Page,Y Addressing

Zero page,Y addressing works like zero page,X addressing, but it can be used with only two mnemonics: LDX and STX. If we didn't have zero page,Y addressing mode, we couldn't use absolute indexed addressing with the instructions LDX and STX—and that's the only reason why we have this addressing mode.

Indirect Addressing

There are two subcategories of indexed addressing: indexed indirect addressing and indirect indexed addressing. Both indexed indirect addressing and indirect indexed addressing are used primarily to look up data stored in tables.

If you think the names of these two addressing modes are confusing, you're not the first one with that complaint. I never could keep them sorted until I came up with a little memory trick to help eliminate the confusion. Here's the trick: Indexed indirect addressing, which has an X in the first word of its name, uses the 6502/8502 chip's X register. Indirect indexed addressing, which doesn't have an X in the first word of its name, uses the 6502/8502's Y register. Now we'll look at each of the Commodore's two indirect addressing modes —beginning with indexed indirect addressing.

Indexed Indirect Addressing

Indexed indirect addressing works in several steps. First, the contents of the X register are added to a zero page address—not to the contents of the address, to the address itself. The result of this calculation must always be another zero page address. When this second address is calculated, the value that it contains—together with the contents of the next address—make up a third address. And that third address is (at last) the address that is finally interpreted as the operand of the statement in question.

An example might help clarify this process. Let's suppose that memory addresses $B0 holds the number $00, memory address $B1 holds the number $80, and the X register holds the number 0. Represented in an easier-to-read form:

```
$B0 = #$00
$B1 = #$80
  X = #$00
```

Now let's suppose you are running a program that contains the indexed indirect instruction LDA ($B0,X). If all of these conditions exist when the computer encounters the instruction LDA ($B0,X), the computer adds the contents of the X register (a 0) to the number $B0. The sum of $B0 and 0 is $B0. So the computer goes to memory address $B0 and $B1. It finds the number $00 in memory address $B0, and the number $80 in $B1.

Because 6502/8502-based computers store 16-bit numbers in reverse order (low byte first) the computer interprets the number found in $B0 and $B1 as $8000. It loads the accumulator with the number $8000, the 16-bit value stored in $B0 and $B1.

Now let's imagine that when the computer encounters the statement LDA ($B0,X), its 6502/8502's X register holds the number 04, instead of the number 00. Here is a chart illustrating these values, plus a few more values that we'll be using shortly:

```
$B0 = #$00
$B1 = #$80
$B2 = #$0D
$B3 = #$FF
$B4 = #$FC
$B5 = #$1C
  X = #$04
```

If these conditions exist when the computer encounters the instruction LDA ($B0,X), the computer adds the number $04 (the value in the X register) to the number $B0, and then goes to memory addresses $B4 and $B5. In these two addresses, it finds the final address (low byte first) of the data it is looking for, in this case, $1CFC.

Indexed indirect addressing is not used in many assembly language programs. When it is used, its purpose is to locate a 16-bit address stored in a table of addresses on page zero. Because space on page zero is so hard to find, it's not very likely that you'll be able to store many data tables there. So, unless you get a job designing operating systems or other kinds of high-performance programs, it's unlikely that you'll ever find much use for indexed indirect addressing.

Indirect Indexed Addressing

Indirect indexed addressing, unlike indexed indirect addressing, is used quite often in assembly language programs. Indirect indexed addressing uses the Y register (never the X register) as an offset to calculate the base address of the start of a table. The starting address of the table has to be stored on page zero, but the table itself doesn't have to be.

When an assembler encounters an indirect indexed address in a program, the first thing it does is PEEK into the zero page address enclosed in the parentheses that precede the Y. The 16-bit value stored in that address and the following address are then added to the contents of the Y register. The value that results is a 16-bit address, the address the statement is looking for.

Here's an example of indirect indexed addressing. Suppose the computer is running a program and comes to the instruction ADC ($B0),Y. It looks into memory addresses $B0 and $B1. $B0 contains the number $00; $B1 contains the number $50; and the Y register contains a 0. Here is a chart that illustrates these conditions:

```
$B0 = #$00
$B1 = #$50
  Y = #$04
```

If these states exist when the computer encounters the instruction ADC ($B0),Y, the computer combines the numbers $00 and $50, and comes up with the address $5000 (in the 6502/8502 chip's peculiar low-byte-first fashion). It then adds the contents of the Y register (4 in this case) to the number $5000, and ends up with a total of $5004. That number—$5004—is the final value of the operand ($B0,Y). So the contents of the accumulator are added to whatever number is stored in memory address $5004.

After you understand indirect indexed addressing, it can become a very valuable tool in assembly language programming. Only one address—the starting address of a table—has to be stored on page zero, where space is always scarce. Yet that address, added to the contents of the Y register, can be used as a pointer to locate any other address in your computer's memory.

Unindexed Indirect Addressing

Unindexed indirect addressing is a special kind of addressing mode that can be used with only one 6502/8502 mnemonic: the JMP instruction. When unindexed indirect addressing is used, a 16-bit number is placed inside the parentheses that follow the JMP instruction. This number serves as a pointer to a pair of memory registers which, together, contain the address to which the desired jump will be made. Let us suppose, for example, that memory address $0C00 contains the value $00 and that address $0C01 holds the value $06. Now suppose that the statement JMP ($0C00) is included in a Commodore 128 assembly language program. If this is the case, the program being executed jumps to the address $0600, and not to the address $0C00, which would be the case if the jump instruction were simply JMP $0C00 without the parentheses.

A Pseudo Address: The Stack

That completes our examination of the thirteen official addressing modes used in 6502/8502 assembly language programming. But as long as we're discussing 6502/8502 addressing modes, I'll introduce a programming tool that's related closely to addressing: a tool called the 8502 stack.

The 8502 stack (often referred to simply as the stack) occupies the 256 bytes of RAM that extend from memory address $100 to memory address $1FF. (A special feature of the C-128 is that the stack, like page zero, can be relocated by the programmer. But we won't go into detail about that until we get to chapter 10.) The stack is what programmers sometimes call a LIFO (last in, first out) block of memory. It is often compared to a spring-loaded stack of plates in a diner; when you put a number in the memory location on top of the stack, it covers up the number that was previously on top. So the number on top of the stack must be removed before you can access the number under it (which was previously on top).

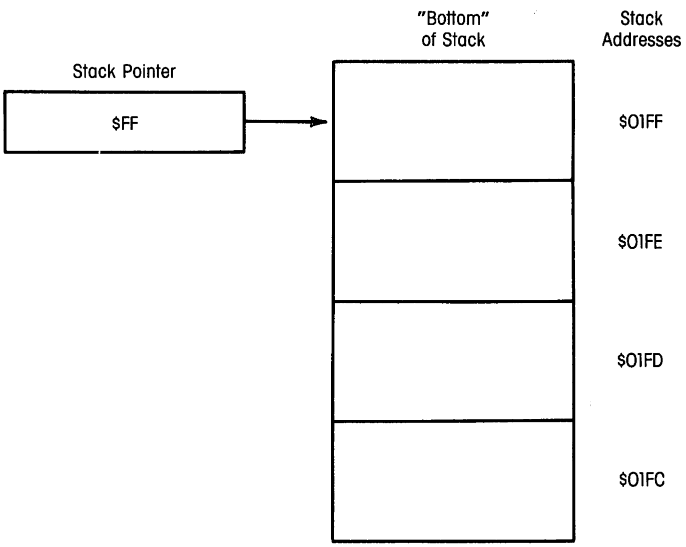
Although the stacked plates comparison is a useful technique for describing how the stack works, it is not completely accurate. Actually, the stack is only a block of RAM, and blocks of RAM don't really move up and down like a stack of plates inside the Commodore 128. When you place a number on the 8502 stack, here's what really happens.

The 8502 stack, as mentioned, is ordinarily situated in a block of memory that extends from memory register $0100 to memory register $01FF. This block of memory is used from high memory downward; that is, the first number stored on the stack is in register $01FF, the next number is placed in register $01FE, and so on. Because of this from-the-top-down storage system, the last stack address that can be used is memory register $0100.

The Commodore 128's 8502 chip keeps track of stack manipulations with the help of a special register called the stack pointer. (The stack pointer was described briefly in chapter 3.) When nothing is stored on the stack, the value of the stack pointer is $FF. Add $100 to that number and you get $01FF—the highest memory address on the stack, and the address that is used for the next (or, in this case, the first) value stored on the stack.

As soon as a value is stored on the stack, the C-128's 8502 chip automatically decrements the stack pointer by one. And each time another value is stored on the stack, the stack pointer is decremented

Figure 6-1
How the stack pointer works



again. Therefore, the stack pointer always points to the address of the *next* value that will be stored on the stack.

Stack Operations

Let us suppose, now, that several numbers are stored on the stack. Let us also suppose that the time has come to retrieve one of those values from the stack. What happens?

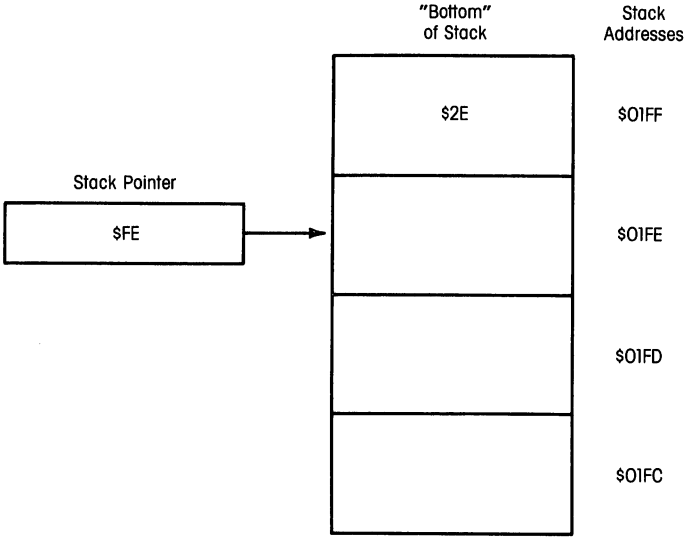
You can probably guess the answer. When we retrieve a number stored on the stack, the value of the stack pointer is incremented by one. This effectively removes one value from the stack because it means that the next value stored on the stack has the same position on the stack as the one that was removed. That's a little tricky to comprehend, given the upside-down nature of the stack. Figure 6-1 might help you understand how this works. This illustration shows an empty stack, with the stack pointer pointing to the first available address on the stack: $01FF.

Now let's place a number (whose value is arbitrary) on the stack. This kind of operation is illustrated in figure 6-2. In this figure, the value of the stack pointer has been decremented, and the number we have placed on the stack is now stored at the highest address in the stack, memory register $01FF.

Figure 6-3 shows what happens if we place another number (also an arbitrary value) on the stack. In figure 6-3, the stack pointer is decremented again, and a second number is now on the stack.

Figure 6-4 shows what happens when we "remove" one number from the stack. In figure 6-4, stack address $01FE still holds the value $B0, but the value of the stack pointer has been incremented and now points to memory address $01FE. So the next number

Figure 6-2
Placing a number on the stack



placed on the stack will be stored at memory address $01FE. When that happens, the number previously stored in that stack position—$B0—is erased.

To see how that works, we'll store one more number on the stack. This time, for no special reason, the value of the number placed on the stack is $17. This process is illustrated in figure 6-5.

As figure 6-5 shows, register $01FE now holds the value $17. The value of the stack pointer has been incremented, the value $B0 has been erased by the value $17, and the next number placed on the stack will be stored in memory register $01FD. And that's how the 8502 stack works in the Commodore 128.

Figure 6-3
Placing another number on the stack

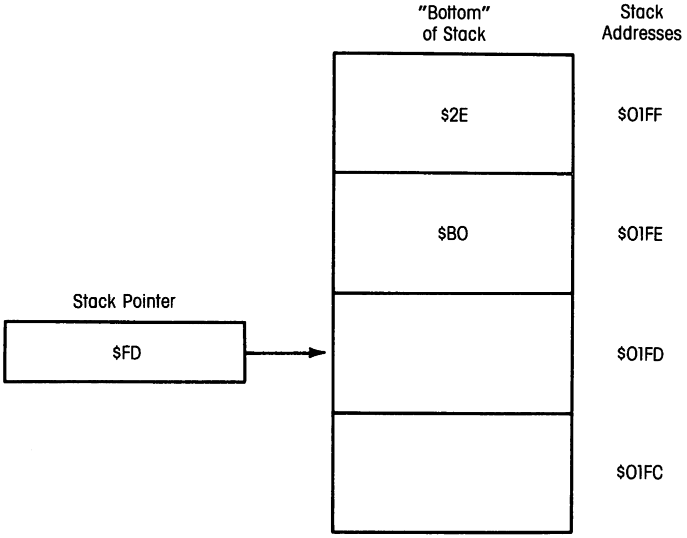


Figure 6-4
Pulling a number off the stack

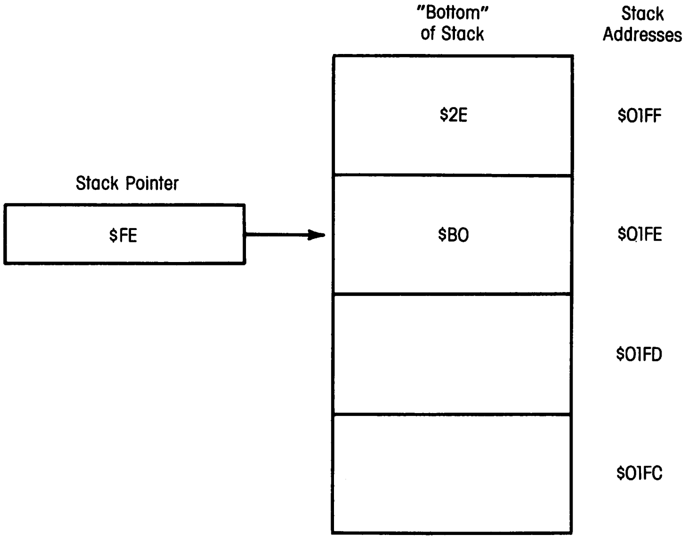
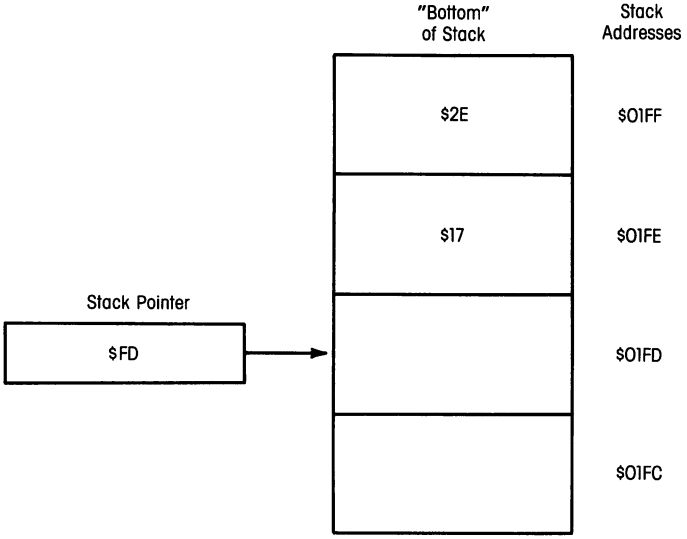


Figure 6-5
One last stack manipulation



Using the Stack in a Program

As previously mentioned, the 6502/8502 processor often uses the stack for temporary data storage during the operation of a program. When a program jumps to a subroutine, for example, the 6502/8502 chip takes the memory address that the program will later have to return to, and pushes that address onto the top of the stack. Then, when the subroutine ends with an RTS instruction, the return address is pulled from the top of the stack and loaded into the 6502/8502's program counter. Then the program can return to the proper address, and normal processing can resume.

The stack is also used quite often in user-written programs.

Listing 6-5 is an example of a routine that makes use of the stack. You may recognize it as a variation on the 8-bit addition program that we've been using throughout this book. The program in listing 6-5 is divided into two parts. In routine 1, we'll put two 8-bit numbers on the stack. In routine 2, we'll take them off the stack and add them. Routine 1 should be typed first. However, before the program is assembled and executed, routine 2 should be appended to routine 1.

Listing 6-5 STACKADD.SRC program

```
1000 ;
1010 ; STACKADD.SRC, ROUTINE 1
1020 ;
1030 *=$1300
1040 ;
1050  LDA #35 ;(OR ANY OTHER 8-BIT NUMBER)
1060  PHA
1070  LDA #49 ;(OR ANY OTHER 8-BIT NUMBER)
1080  PHA
1100 ;
1110 ; STACKADD.SRC, ROUTINE 2
1120 ;
1130 ; WHEN THIS PROGRAM BEGINS, TWO
1140 ; 8-BIT NUMBERS ARE ON THE STACK
1150 ;
1160  CLD
1170  CLC
1180  PLA
1190  STA $FA
1200  PLA
1210  ADC $FA
1220  STA $FB
1230  RTS
```

Listing 6-6 shows what happens when we append routine 2 to routine 1.

Listing 6-6 STACKADD2.SRC program

```
1000 ;
1010 ; STACKADD2.SRC
1020 ;
1030 *=$1300
1040 ;
1041 ; ROUTINE 1
1042 ;
1050  LDA #35 ;(OR ANY OTHER 8-BIT NUMBER)
1060  PHA
1070  LDA #49 ;(OR ANY OTHER 8-BIT NUMBER)
1080  PHA
1100 ;
1130 ; ROUTINE 2
1150 ;
1160  CLD
```

Listing 6-6 cont.

```
1170   CLC
1180   PLA
1190   STA $FA
1200   PLA
1210   ADC $FA
1220   STA $FB
1230   RTS
```

Listing 6-6, a simple, straightforward 8-bit addition routine, shows how easy and convenient it can be to use the stack in assembly language programs. In line 1180, a value is pulled from the stack and stored in the accumulator. Then, in line 1190, the value is stored in memory address $FA. In lines 1200 and 1210, another value is pulled from the stack and added to the value now stored in $FA. The result of this calculation is then stored in $FB, and the routine ends.

As you can see, the stack can be a very convenient place to store data temporarily. The stack can also be a very memory efficient tool because it doesn't require the use of dedicated storage registers. It can also save time, because it takes only one instruction to push a value onto the stack and only one instruction to retrieve a value that has been stored there.

An Important Warning

But beware: It can very dangerous for beginning programmers to play with the stack. When you use the stack in an assembly language routine, it's extremely important to leave the stack exactly as you found it when the routine ends. If you've placed a value on the stack during a routine, it must be removed from the stack before the routine ends and normal processing resumes. Otherwise, there might be "garbage" on the stack when the next routine is called, and that can result in program crashes, memory wipeouts, and various other programming disasters. Remember: Mismanagement of the hardware stack is extremely hazardous to the health of assembly language programs! So, if you take care to manage the stack properly, in other words, if you make sure to clear the stack after each use, it can be a very powerful programming tool.

Mnemonics that make use of the stack are PHA ("push the contents of the accumulator onto the stack"), PLA ("pull the top value off the stack and store it in the accumulator"), PHP ("push the contents of the P register onto the stack"), and PLP ("pull the top value off the stack and store it in the P register").

The PHP and PLP operations are often included in assembly language subroutines so that the contents of the P register won't be wiped out during subroutines. When you jump to a subroutine that may change the status of the P register, it's always a good idea to start the subroutine by pushing the contents of the P register onto the stack. Then, just before the subroutine ends, you can restore the P register's previous state with a PHP instruction. In that way, the P register's contents won't be destroyed during the subroutine.

7

For a Loop

Looping and branching in C-128 assembly language

Now we're going to start having some real fun with Commodore 128 assembly language. In this chapter, you'll learn how to print messages on the C-128 screen, how to use ASCII characters in assembly language programs, and how to perform a number of other neat tricks in 8502 assembly language. We're going to accomplish these feats using some new and fairly advanced assembly language programming techniques, along with a few not-so-new variations that were covered in earlier chapters.

These are some of the new programming techniques discussed in this chapter:

- incorporating data tables into assembly language programs
- incorporating loops into assembly language instructions
- programming loops using comparison and branching instructions

In the programs used to illustrate these techniques, we'll make extensive use of the BSOUT kernel routine, which can be accessed at call address $FFD2. BSOUT—which was called CHROUT in the Commodore 64 kernel—is the routine we used in chapter 5 to print the character X on your computer screen. We'll use the BSOUT routine in this chapter to print some short messages on the C-128 screen. In later chapters, the same routine is used in some fancier applications.

Q.S Program

We'll start with listing 7-1, a program called Q.S. It is written on a Commodore 128 using the Merlin 128 assembler/editor system. Although minor modifications might be needed, it can be typed and assembled using any assembler compatible with the Commodore 128 or the Commodore 64.

Listing 7-1
Q.S program

```
*
* Q.S
*
          ORG   $1300
*
BUFLEN    EQU   23
BSOUT     EQU   $FFD2
*
          JMP   BEGIN
*
TEXT      DFB   87,72,69,82,69,32,73,83
          DFB   32,84,72,69,32,67,79,77
          DFB   77,79,68,79,82,69,63
*
```

Listing 7-1 cont.

```
15 BEGIN      LDX    #0
16 *
17 LOOP       LDA    TEXT,X
18            JSR    BSOUT
19            INX
20            CPX    #BUFLEN
21            BNE    LOOP
22            RTS
```

When you type, assemble, and execute the program, it will print a cryptic message on your video screen.

Running the Program

When you've finished typing the Q.S program, it would be a good idea to assemble it and then save it on a disk in both its source code and object code versions. After the program is safely stored on a disk, you can run it using any of the three methods covered in chapter 5: from BASIC, from DOS, or using the C-128 machine language monitor.

When you run the program, you'll see what it does. But how does it do it? Let's start with an explanation of the pseudo op code DFB, which appears in lines 11 through 13. (As mentioned in chapter 4, pseudo ops, or directives, vary from assembler to assembler, so you might have to use a .BYT or .BYTE directive instead of DFB if you have a Commodore or TSDS assembler.)

When you use a DFB directive (or one of its equivalents) in a program, the bytes that follow the directive are assembled into consecutive locations in RAM. In the Q.S program, the bytes that follow the label TEXT are ASCII codes for a series of text characters—as you can see when you run the program.

Looping the Loop

As explained in chapter 6, the X and Y registers in the 6502/8502 chip can be progressively incremented and decremented during loops in a program. In the Q.S program, the X register is incremented from 0 to 23 during a loop in which the program reads characters in a text string. The characters are written as ASCII codes in lines 11 through 13. In line 15, the statement LDX #0 loads the X register with a 0. Then, in line 17, the loop begins.

The first statement in the loop is LDA TEXT,X. Each time the loop cycles, this statement uses indexed addressing to load the accumulator with an ASCII code for a text character. Then, in line 18, the BSOUT kernel routine prints each character on the screen. By the time the loop ends, all 23 characters in lines 11 through 13 have been printed on the screen.

The first time the program reaches line 17, there is a 0 in the X register (because we've just loaded a 0 into the X register). So the first time the program encounters the statement LDA TEXT,X, the accumulator is loaded with the hexadecimal number $54, the ASCII code for the letter T. So, in line 18, the BSOUT kernel routine takes the hex number $54 from the accumulator, recognizes it as the ASCII code for a T, and prints a T on the screen.

(Incidentally, we don't need # symbols before the numbers in lines 11-13 because numbers that follow the DFB pseudo op and its variants are automatically interpreted as literal numbers.)

Now let's move on to line 19. The mnemonic you see there—INX—means "increment the X register." The first time the program makes its way through the loop that starts at line 17, the X register holds a 0. But as soon as the BSOUT routine has printed its first character on the screen, the INX instruction in line 19 increments that 0 to a 1.

Next, in line 20, is the instruction CPX #BUFLEN. Look back at line 6, and you'll see that BUFLEN is a constant that equates to the number 23. So the instruction CPX #BUFLEN means "compare the value in the X register to the literal number 23." We perform this comparison to determine whether 23 characters have been printed on the screen. There are 23 characters in the text string that we're printing, and when we've printed all of them, we want to print a carriage return and end our program.

Comparison Instructions

There are three comparison instructions in 6502/8502 assembly language: CMP, CPX, and CPY. CMP means "compare to a value in the accumulator." When you use the instruction CMP, followed by an operand, the value expressed by the operand is subtracted from the value in the accumulator. This subtraction operation is not performed to determine the exact difference between these two values; it merely determines whether or not they are equal and, if they are not equal, which one is larger.

If the value in the accumulator is equal to the tested value, the zero flag of the processor status register is set to 1. If the value in the accumulator is not equal to the tested value, the zero flag is left in a cleared state. If the value in the accumulator is less than the tested value, then the carry flag of the processor status register is left in a clear state. And if the value in the accumulator is greater than or equal to the tested value, the zero flag is set to 1 and the carry flag is set.

CPX and CPY work like CMP, except they are used to compare values with the contents of the X and Y registers. They have the same effects that CMP has on the status flags of the processor status register.

Conditional Branching Instructions

The three comparison instructions in Commodore 128 assembly language are usually used with eight other assembly language instructions—the eight conditional branching instructions mentioned in chapter 6.

The Q.S program in listing 7-1 contains a conditional branching instruction in line 21. That instruction is BNE LOOP, which means "branch to the statement labeled LOOP if the zero flag (of the processor status register) is not set." This instruction uses a confusing convention of the 8502 chip. In the 8502's processor status register, the zero flag is set (equals 1) if the result of an operation is 0. The zero flag is cleared (equals 0) if the result of an operation is not zero.

This is all academic, however, as far as the result of the statement BNE LOOP is concerned. When your Commodore 128 encounters the BNE LOOP instruction in line 21, it keeps branching back to line 17 (the line labeled LOOP) as long as the value of the X register is not decremented to zero.

After the value of the X register is decremented to zero, the statement BNE LOOP in line 21 is ignored, and the program moves to the next line. And that line contains an RTS instruction that ends the program.

Now let's take a closer look at conditional branching instructions. As you may recall from chapter 6, there are eight conditional branching instructions in 6502/8502 assembly language. They all begin with the letter B, and they're also called relative addressing instructions, or branching instructions. These eight instructions, and their meanings, are listed in table 7-1.

Table 7-1
8502 Branching Instructions

Instruction	Meaning
BCC	Branch if the carry (C) flag of the processor status (P) register is clear. If the carry flag is set, the operation will have no effect.
BCS	Branch if the carry (C) flag is set. If the carry flag is clear, the operation will have no effect.
BEQ	Branch if the result of an operation is zero, if the zero (Z) flag is set.
BMI	Branch on minus if an operation results in a set negative (N) flag.
BNE	Branch if not equal to zero, if the zero (Z) flag isn't set.
BPL	Branch on plus if an operation results in a cleared negative (N) flag.
BVC	Branch if the overflow (V) flag is clear.
BVS	Branch if overflow (V) flag is set.

How Branching Differs from Jumping

In a few moments, we'll use some of these instructions in another assembly language program. First, though, this is a good time to point out some important differences between the branching instructions listed in table 7-1 and another category of 6502/6510/8502 instructions: jump instructions, which were mentioned earlier in this book.

There are two jump instructions in 6502 assembly language: JMP and JSR. As you may recall from previous chapters, the JMP mnemonic is used much like the GOTO instruction in BASIC; when a JMP instruction is encountered in an assembly language program, the program jumps to whatever memory address is specified by the operand that follows the JMP instruction.

The assembly language instruction JSR is used much like BASIC's GOSUB instruction. When a JSR instruction is encountered in an assembly language program, the memory address of the next instruction in the program is stored on the hardware stack. Then, the program jumps to whatever memory address is specified by the operand that follows the JSR instruction.

The mnemonic JSR is designed primarily for use with subroutines. In 6502/6510/8502 assembly language, a subroutine almost always ends with an RTS instruction. In assembly language, RTS is the opposite of JSR. When an RTS instruction is encountered in a program, a memory address is removed from the stack, and processing immediately jumps to that address. If the RTS instruction is used to end a subroutine, then the address pulled from the stack is ordinarily the one placed there by the JSR instruction that invoked the subroutine. This means processing of the program resumes where it left off: at the line following the JSR instruction that invoked the subroutine.

We have already observed one difference between branching instructions and the jump instructions JMP and JSR: branching instructions are conditional and jump instructions are unconditional. When a jump instruction is encountered in a program, it is always carried out. But when a branching instruction is encountered in a program, it is carried out only if specific conditions are fulfilled.

There is also another important difference between a jumping instruction and a branching instruction. In machine language, the operand that follows a jump instruction is always expressed as a 2-byte value and is always interpreted as the actual starting address of the destination of the jump instruction. But when a branching instruction is assembled into machine language, the operand that follows the branching instruction is converted to a signed 1-byte number. When the program is executed, this signed 1-byte number is interpreted as an offset that points to the starting address of the destination of the branch instruction.

This all sounds quite complicated, but some simple examples should make it clearer. Let's start with a sample statement containing a jump instruction:

```
JMP $C000
```

If this statement is assembled into machine language and executed, the result is quite straightforward; the value $C000 is loaded into the computer's program counter, and a jump to memory address $C000 takes place.

A Branching Program

Unfortunately, branching instructions are more complicated than jump instructions. Listing 7-2 is a program that uses the BCC branching instruction, which means "branch if carry set." The program is called BRANCHIT.S, for obvious reasons.

Listing 7-2
BRANCHIT.S program (source code version)

```
1             ORG    $1300
2 *
3 WHAZIS      EQU    $0C00
4 *
5             LDA    #5
6             CLC
7             ADC    WHAZIS
8             BCS    RETURN
9             TAX
10 RETURN     RTS
```

BRANCHIT.S is a very straightforward little program. In line 5, the literal number 5 is loaded into the accumulator. Then the 6502/8502 carry flag is cleared, and the value stored in memory address $0C00 (which is labeled WHAZIS) is added to the value stored in the accumulator (now 5). Next, in line 8, a branching instruction is invoked. If adding 5 to the value of WHAZIS results in a carry—that is, if the sum of 5 and WHAZIS is greater than 255—then the routine branches to line 10 and ends. But if the sum of 5 and WHAZIS does not result in a carry—that is, if the sum is less than 255 —then the sum is transferred to the X register before the routine ends.

Listing 7-3 is an assembled listing of the BRANCHIT.S program.

Listing 7-3
BRANCHIT.S program (object code version)

```
                   1                 ORG    $1300
                   2     *
                   3     WHAZIS      EQU    $0C00
                   4     *
1300: A9 05        5                 LDA    #5
1302: 18           6                 CLC
1303: 6D 00 0C     7                 ADC    WHAZIS
1306: B0 01        8                 BCS    RETURN
1308: AA           9                 TAX
1309: 60           10    RETURN      RTS

--End Assembly, 10 bytes, Errors: 0
```

Listing 7-3 cont.

```
Symbol table - alphabetical order:

  RETURN   =$1309      WHAZIS   =$0C00


Symbol table - numerical order:

  WHAZIS   =$0C00      RETURN =$1309
```

Carefully examine lines 8 through 10 of listing 7-3, and you'll see how the branching instruction in the BRANCHIT.S program works. In line 8, to the left of the line number, these figures appear:

```
1306:B0 01
```

The first figure in this line—1306—is the memory address where the BCS instruction is stored when it is assembled into machine language. The second figure in the line—B0—is the machine language equivalent of the BCS instruction. The third number—01—is an offset value that must be computed by the Commodore 128's 8502 chip before it can carry out the BCS instruction.

What Are Offset Values?

And what, exactly, is an offset value? Well, in a 6502/8502 branching instruction, an offset value is a signed number added to a given memory address to compute the destination address of the branching instruction. The address to which it must be added is always the address that *follows* the statement containing the branching instruction. Therefore, the offset in line 8 of the BRANCHIT.S program is 1. When that 1 is added to the address of the instruction following the branching instruction—$1308—the sum is $1309. And that is the address of the RTS instruction that ends the BRANCHIT.S program.

Following are more details about how this works. When you write a branching instruction in assembly language, you can follow it with either a literal address or a label that equates to an address. But then, when your program is assembled into machine language, the assembler converts the literal address or label into an offset value. From then on, each time the 8502 chip encounters a branching instruction during the execution of the assembled program, it automatically uses the offset that follows each branching instruction to compute the destination address of the branch.

Another fact that's important to remember is that an offset that follows a branching instruction can never be more than one byte. And, because this one byte is always interpreted by the 6502/8502 chip as a signed number, a branching offset can't be smaller than −128 and larger than +127. Because this displacement is always added to the address of the first instruction that follows a branching instruction, the effective displacement of a branching instruction is only between −126 and +129. So branches are subject to certain length limitations; specifically, the destination address of a branching instruction cannot be more than 126 bytes lower than, or more than 129 bytes higher than, the address of the first instruction following the branching instruction.

But what if you want to write an instruction that branches to an address not within these limitations? Well, that isn't too difficult. If you want to exceed the distance limitations of a branching instruction, have the branching instruction branch to a jumping instruction, which has no such restrictions. Listing 7-4 shows how to do it.

Listing 7-4
BRANCHIT.S program with a jump instruction

```
1               ORG    $1300
2  *
3  WHAZIS       EQU    $0C00
4  *
5                LDA    #5
6               CLC
7               ADC    WHAZIS
8               BCC    CONT
9               JMP    FARJUMP        ;(CAN BE
10 CONT         TAX                   ;ANYWHERE IN
11              RTS                   ;MEMORY)
```

Something Fancy

As you can see, a very neat trick has been used to overcome the distance limitations of a branching instruction. In this version of the BRANCHIT.S program, the BCS instruction that appeared in the original program is replaced by a BCC instruction. And a new line, containing a JMP instruction that can jump to any address in memory, is inserted following the line containing the BCC instruction. In this version of the program, if adding 5 to the value of WHAZIS results in a carry, the program jumps to an address labeled FARJUMP, which can be situated anywhere. Otherwise, the program jumps to line 10, labeled CONT (for "continue"), and proceeds as before.

Using Branching Instructions

As you may have noticed from the programming examples provided so far in this chapter, the usual way to use a conditional branching instruction in 6502/8502 assembly language is as follows: First, load the X or Y register with a zero or some other value, and then load the A register (or a memory register) with a value to be used for a comparison. Then you use a conditional branching instruction to tell the computer what P register flags to test, and what to do if these tests succeed or fail.

This all sounds very complicated, and, until you get the hang of it, it may be. But after you understand the general concept of conditional branching, you can use a simple table for writing conditional branching instructions, as shown in table 7-2.

Table 7-2
8502 Conditional Branching Instructions

To Test For:	Do This:	And Then This:
A = VALUE	CMP #VALUE	BEQ
A <> VALUE	CMP #VALUE	BNE
A ≥ VALUE	CMP #VALUE	BCS
A > VALUE	CMP #VALUE	BEQ and then BCS
A < VALUE	CMP #VALUE	BCC
A = (ADDR)	CMP $ADDR	BEQ
A <> (ADDR)	CMP $ADDR	BNE
A ≥ (ADDR)	CMP $ADDR	BCS
A > (ADDR)	CMP $ADDR	BEQ and then BCS
A < (ADDR)	CMP $ADDR	BCC
X = VALUE	CPX #VALUE	BEQ
X <> VALUE	CPX #VALUE	BNE
X ≥ VALUE	CPX #VALUE	BCS
X > VALUE	CPX #VALUE	BEQ and then BCS
X < VALUE	CPX #VALUE	BCC
X = (ADDR)	CPX $ADDR	BEQ
X <> (ADDR)	CPX $ADDR	BNE
X ≥ (ADDR)	CPX $ADDR	BCS
X > (ADDR)	CPX $ADDR	BEQ and then BCS
X < (ADDR)	CPX $ADDR	BCC
Y = VALUE	CPY #VALUE	BEQ
Y <> VALUE	CPY #VALUE	BNE
Y ≥ VALUE	CPY #VALUE	BCS
Y > VALUE	CPY #VALUE	BEQ and then BCS
Y < VALUE	CPY #VALUE	BCC
Y = (ADDR)	CPY $ADDR	BEQ
Y <> (ADDR)	CPY $ADDR	BNE
Y ≥ (ADDR)	CPY $ADDR	BCS
Y > (ADDR)	CPY $ADDR	BEQ and then BCS
Y < (ADDR)	CPY $ADDR	BCC

In 6502/8502 assembly language, comparison instructions and conditional branch instructions are usually used together. In the Q.S program in listing 7-1, for example, the comparison instruction CPX and the branch instruction BNE are used together in a loop controlled by incrementing a value in the X register. For each loop cycle, the value in the X register is progressively incremented or decremented. And each time the program comes to line 20, the value in the X register is compared to the literal number 23. When the value in the X register equals 23, the loop ends. Therefore, the program keeps looping back to line 17 until 23 characters are printed on the screen.

Got it? Good! Then we're ready to make some improvements in the Q.S program. These improvements are incorporated into a new program, called A.S, which appears in listing 7-5.

Listing 7-5
A.S program

```
*
* A.S
*
         ORG  $1300
EOL      EQU  13
BUFLEN   EQU  24
FILLCH   EQU  $20
BSOUT    EQU  $FFD2
*
         JMP  START
*
TEXT     ASC  'HE HAS MOVED UP TO 128K'
         DFB  13
*
* CLEAR TEXT BUFFER
*
START    LDA  #FILLCH
         LDX  #BUFLEN
STUFF    DEX
         STA  TXTBUF,X
         BNE  STUFF
*
* STORE MESSAGE IN BUFFER
*
         LDX  #0
LOOP1    LDA  TEXT,X
         STA  TXTBUF,X
         CMP  #EOL
         BEQ  PRINT
         INX
         CPX  #BUFLEN
         BCC  LOOP1
*
* PRINT MESSAGE
*
PRINT    LDX  #0
LOOP2    LDA  TXTBUF,X
         PHA
         JSR  BSOUT
         PLA
         CMP  #EOL
         BNE  NEXT
         JMP  FINI
NEXT     INX
         CPX  #BUFLEN
         BCC  LOOP2
*
FINI     RTS
*
TXTBUF   DS   BUFLEN
         END
```

The A.S program, which is quite similar to the Q.S program, is also written using a Merlin 128 assembler. But with a few modifications you can type, assemble, and run it on any assembler/editor compatible with the Commodore 128 or the Commodore 64.

Now, if you like, type, assemble, and save the A.S program. Then we'll discuss the differences between it and its predecessor.

Text Strings

The most obvious difference between the Q.S and A.S programs is the way they handle text strings. In the Q.S program, we used a text string made up of ASCII codes. There's also a text string in A.S, but it's made up of actual characters. Because of that difference, A.S is a much easier program to write than Q.S—and it's much easier to read and understand.

Another important difference between A.S and its predecessor is the way we write the loop that reads the characters. In Q.S, the loop counts the number of characters that have been printed on the screen, and ends when the count reaches 23. That's a perfectly good system—for printing text strings that contain 23 characters. Unfortunately, it isn't so great for printing strings of other lengths. So it isn't a very versatile routine for printing characters on a screen.

Testing for a Carriage Return

A.S is more versatile than Q.S because it can print strings of almost any length on a screen. That's because the A.S program doesn't keep track of the number of characters it has printed by maintaining a running count of how many letters have been printed. When the program encounters a character, it tests the character to see whether its value is $0D—the ASCII code for a carriage return, or end-of-line (EOL), character. If the character is not an EOL, the computer prints it on the screen and goes to the next character in the string. If the character is an EOL character, the computer prints a carriage return on the screen and the routine ends.

Using a Text Buffer

Another difference between Q.S and A.S is that the latter program doesn't read a character and print it on the screen in the same step. Instead, the characters are placed in a buffer, and then the contents of the buffer are printed on the screen.

Text buffers are often used in assembly language programs because they are both versatile and easy to use. Text can be loaded into a buffer in many ways: from a keyboard, from a telephone modem, or even directly from a computer's memory. After a string is in a buffer, it can be removed from the buffer in just as many different ways—no matter how the characters got into the buffer, and no matter what characters they are. So, after you write a few subroutines that fill and then process a buffer in some manner, those subroutines can be used for many different purposes. A buffer can therefore serve as a central repository for text strings, which are then easily accessible in many different ways.

Before you use a text buffer, though, it's always a good idea to clear it; otherwise, it might be cluttered with leftover characters. So a buffer-clearing routine is included in the A.S program. It's a short and simple routine, but it does the job. It will clear a text buffer—or any other block of memory that doesn't exceed its length limitations—and fill the buffer with spaces, zeros, or any other value you might choose. In the A.S program, the routine fills the buffer with a string of spaces, which appear as blank spaces on your computer screen.

As you continue to work with assembly language, you'll find that memory-clearing routines such as this one can come in very handy in different kinds of programs. Word processors, telecommunications programs, and many other kinds of software packages make extensive use of routines that clear values from blocks of memory and replace them with other values.

The memory-clearing routine in the A.S program is not very complicated. Using indirect addressing and an X register countdown, it fills each memory address in a text buffer (TXTBUF) with a designated "fill character" (FILLCH). Then the program ends.

The buffer-clearing routine in A.S works with any 8-bit fill character, and with any buffer length (BUFLEN) up to 255 characters. Later in this book, you'll find some 16-bit routines that can fill longer blocks of RAM with values.

Name Game Program

The final program in this chapter, in listing 7-6, uses many of the programming techniques mentioned so far. The program is called the NAME GAME, and I wrote it in BASIC years ago. The version that appears in listing 7-6 is written on a Commodore 128 using a Merlin 128 assembler. But, if you own a Commodore or TSDS assembler, you shouldn't have much trouble converting it into source code that's compatible with your assembler/editor.

Listing 7-6
NAME GAME
program

```
*
* NAME GAME
*
        ORG   $1300
*
EOL     EQU   $0D          ;RETURN
EOF     EQU   $03          ;EOF CHR
FILLCH  EQU   $20          ;SPACE
BUFLEN  EQU   40
BASIN   EQU   $FFCF
```

Listing 7-6 cont.

```
BSOUT    EQU   $FFD2
TEMPTR   EQU   $FB
*
         JMP   START
*
TXTBUF   DS    40
*
TITLE    ASC   'THE NAME GAME'
         HEX   0D
HELLO    ASC   'HELLO, '
         HEX   03
QUERY    ASC   'WHAT IS YOUR NAME?'
         HEX   0D
NAME     ASC   'GEORGE'
         HEX   0D
REBUFF   ASC   'GO AWAY,'
         HEX   03
DEMAND   ASC   'BRING ME GEORGE!'
         HEX   0D
GREET    ASC   'HI, GEORGE!'
         HEX   0D
*
* CLEAR TEXT BUFFER
*
FILL     LDA   #FILLCH
         LDX   #BUFLEN
DOFILL   DEX
         STA   TXTBUF,X
         BNE   DOFILL
         RTS
*
PRINT    LDY   #0
SHOW     LDA   (TEMPTR),Y
         CMP   #EOF
         BEQ   DONE
         PHA
         JSR   BSOUT
         PLA
         CMP   #EOL
         BNE   NEXT
         JMP   DONE
NEXT     INY
         CPY   #BUFLEN
         BCC   SHOW
DONE     RTS
*
* PRINT 'THE NAME GAME'
*
START    LDA   #EOL
```

Listing 7-6 cont.

```
          JSR   BSOUT
          LDA   #<TITLE
          STA   TEMPTR
          LDA   #>TITLE
          STA   TEMPTR+1
          JSR   PRINT
          LDA   #EOL
          JSR   BSOUT

*
* PRINT 'HELLO ...'
*
          LDA   #<HELLO
          STA   TEMPTR
          LDA   #>HELLO
          STA   TEMPTR+1
          JSR   PRINT
*
* PRINT 'WHAT IS YOUR NAME?'
*
ASK       LDA   #<QUERY
          STA   TEMPTR
          LDA   #>QUERY
          STA   TEMPTR+1
          JSR   PRINT
          LDA   #EOL
          JSR   BSOUT
*
* INPUT A TYPED LINE
*
          JSR   FILL
          LDX   #0
KEY       JSR   BASIN
          STA   TXTBUF,X
          CMP   #EOL
          BEQ   COMPARE
          INX
          JMP   KEY
*
* IS THE NAME 'GEORGE'?
*
COMPARE   JSR   BSOUT           ;PRINT RETURN
          LDX   #0
CHECK     LDA   TXTBUF,X
          CMP   NAME,X
          BNE   NOGOOD
          CMP   #EOL
          BEQ   DUNIT
          INX
```

Listing 7-6 cont.

```
           CPX    #BUFLEN
           BCS    DUNIT
           JMP    CHECK
*
* NO; PRINT 'GO AWAY ...'
*
NOGOOD     LDA    #EOL
           JSR    BSOUT
           LDA    #<REBUFF
           STA    TEMPTR
           LDA    #>REBUFF
           STA    TEMPTR+1
           JSR    PRINT
*
* PRINT PLAYER'S NAME
*
           LDA    #<TXTBUF
           STA    TEMPTR
           LDA    #>TXTBUF
           STA    TEMPTR+1
           JSR    PRINT
           LDA    #EOL
           JSR    BSOUT
*
* PRINT 'BRING ME GEORGE!'
*
           LDA    #<DEMAND
           STA    TEMPTR
           LDA    #>DEMAND
           STA    TEMPTR+1
           JSR    PRINT
           LDA    #EOL
           JSR    BSOUT
           JMP    ASK
*
* YES; PRINT GREETING
*
DUNIT      LDA    #EOL
           JSR    BSOUT
           LDA    #<GREET
           STA    TEMPTR
           LDA    #>GREET
           STA    TEMPTR+1
           JSR    PRINT
           RTS
```

If you've typed, assembled, and executed the programs called Q.S and A.S, you shouldn't have much trouble understanding how the NAME GAME works. Using several fairly simple subroutines, it prints a short message on your screen and then waits for you to type a response. If you type a response that the program considers incorrect, it prompts you to try again. When you finally enter the line the program is looking for, you get a "reward" message and the program ends.

How the Program Works

In addition to the kernel routine BSOUT, which is often used in Commodore programs to print characters on a computer screen, the NAME GAME uses another kernel routine—called BASIN—that can read characters typed from a computer keyboard. The call address of the BASIN routine is $FFCF. Detailed instructions on how it is used are in the *Commodore 128 Programmer's Reference Guide.*

BASIN, BSOUT, and other constants are defined in lines 6 through 12 of the NAME GAME program. Then, in line 14, there is a jump instruction that causes execution of the program to start at line 59.

Before the program begins, some space is set aside for a text buffer (in line 16), and the lines of text that are used in the NAME GAME are listed as strings of data in lines 18 through 31. Next, there are two subroutines that are used later in the program. One, labeled FILL, clears the text buffer whenever it's called. The other subroutine, called PRINT, uses the Commodore BSOUT routine to print messages on the screen.

As you type, assemble, and run the NAME GAME, you may notice that it uses its text buffer for lines that are typed at the keyboard, not for lines that are called from RAM. Some kind of buffer is needed for typed lines because the computer must hold them in memory until it does some comparison and printing operations. Another text buffer could have been set up for the lines stored in RAM, but it would have accomplished no real purpose except to use more memory and processing time.

Another part of the NAME GAME program that may be worth special mention is the technique used for storing 16-bit numbers in high-order and low-order 8-bit memory locations. The technique first appears in lines 72 through 75:

```
72   LDA #<HELLO
73   STA TEMPTR
74   LDA #>HELLO
75   STA TEMPTR+1
```

You can probably figure out what this sequence does without too much difficulty. In source code produced by the Merlin 128 assembler and the Commodore 64 assembler, the string #<HELLO means "the low byte of the address labeled HELLO" and the string #>HELLO means "the high byte of the address labeled HELLO." (These strings don't mean the contents of the address, by the way, but the address itself, because in 6502/6510/8502 assembly language, the # symbol identifies a literal number.) So here's what the preceding sequence of code does: First it stores the low-order byte of the 16-bit address of the string HELLO into the memory location labeled TEMPTR (which, as you can see by looking at the program's symbol table, is memory address $FB). Then it stores the high-order byte of the address of the string labeled HELLO into memory location TEMPTR+1, or $FC.

Running the Program

The main part of the NAME GAME program starts at line 57 with a routine that prints the program's title on the screen. The next two routines print the line "HELLO, WHAT IS YOUR NAME?"

After this question is asked, the program clears the text buffer and waits for the player to type a response. As the player types an answer, each typed character is placed in the text buffer. That's all that happens until the player stops typing characters and presses the Return key.

After an EOL character is typed, the program examines the characters, which are stored in the text buffer, to see if they spell the name GEORGE. If the player has not typed the name GEORGE, the computer prints "GO AWAY, *typed name,* BRING ME GEORGE!" (where *typed name* is, logically enough, the name the player typed). Then the game starts again. This process continues until the player weakens and types the name GEORGE. Then the computer prints "HI, GEORGE," and the game ends.

After you've played the NAME GAME two or three times, you'll probably get bored and not want to play it any more. But that's okay, because then we'll both be ready to end this chapter and move to chapter 8.

'Bye, George!

8

Programming Bit by Bit

Performing single-bit operations on binary numbers

One of the most important features of assembly language is its ability to handle operations on individual bits in a binary number. As we've seen in previous chapters, a computer is really nothing but a huge collection of microscopic electronic switches, each one of which is always in one of two states: off or on. In machine language, each of these tiny switches can be expressed as a *binary digit,* or *bit*—in other words, as a 0 (off) or a 1 (on).

In most programming languages, the programmer is not usually concerned with the bits that are strung together inside a computer to form programs and data. Most languages, to make the programmer's job easier, combine bits into groups of eight called *bytes,* or, even more commonly, into even longer groupings called *words.* So programmers who work in high-level languages rarely find it necessary—or even possible—to perform single-bit operations on binary numbers.

But manipulations involving individual bits are a very important part of almost every assembly language program. How important? Well, consider the following facts. The Commodore 128 has 128K—or 131,072 bytes—of RAM. Because there are eight bits to a byte, this means that an off-the-shelf C-128 contains 1,048,576 tiny electronic switches, or bits. When you know how to perform single-bit manipulations on binary numbers, you can individually control each one of the million-plus switches your computer contains. That's a tremendous amount of control to exercise over a computer in any language!

But just what does that mean? After you have control over each bit in your computer's memory, what can you do with it? That's the question we'll try to answer in this chapter.

Back in chapter 1, we learned how to control one of the most important bits in your Commodore's central microprocessor: the carry bit of the 8502 processor status register. Manipulating the processor status register's carry bit is one of the most important bit-manipulation techniques in 8502 assembly language. You've already had considerable experience in using the carry bit in addition programs. In this chapter, you'll have an opportunity to teach your computer how to perform some new tricks using the carry bit of its 8502 processor status register.

Bit Shifting and the Carry Bit

As we've seen a number of times, the Commodore's 8502 microprocessor is an 8-bit chip; it cannot perform operations on numbers larger than 255 without putting them through some fairly tortuous procedures. To process numbers that are larger than 255, the 8502 must split them into 8-bit groups, and then perform the requested operations on each group. Then each number that has been split must be joined. When you become more familiar with this process, it isn't nearly as difficult as it sounds. In fact the electronic "scissors" used in all of this electronic "cutting and pasting" are contained in one tiny bit—the carry bit in the 8502's processor status register.

You've seen how carry operations work in several programs in this book. But to get a clearer look at how the carry works in 8502 arithmetic, it is useful to examine four specialized machine language instructions: ASL (arithmetic shift left), LSR (logical shift right), ROL (rotate left), and ROR (rotate right). These four instructions are used extensively in 8502 assembly language. We'll look at them one at a time, starting with the ASL instruction.

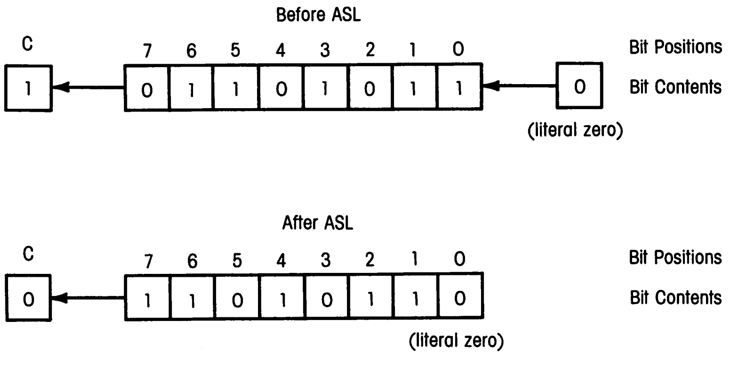
ASL (Arithmetic Shift Left)

As you may be able to figure out from the information we covered in chapter 2, in binary arithmetic, the value of a round binary number is always equal to twice the square of the preceding round binary number. For example, 1000 0000 ($80) is double the number 0100 0000 ($40), which is double the number 0010 0000 ($20), which is double the number 0001 0000 ($10), and so on.

Because of this phenomenon, it is quite easy to multiply a binary number by 2 using bit-manipulation techniques. Just shift every bit in the number to the left one space, and place a 0 in the bit that is emptied by this shift—bit 0, or the rightmost bit of the number. If bit 7 (the leftmost bit) of the number to be doubled is a 1, then provision must be made for a carry.

The entire operation—shifting a bit to the left, with a carry—can be performed by a single instruction in 8502 assembly language. That instruction is ASL, which means "arithmetic shift left." Figure 8-1 shows how the ASL instruction works.

Figure 8-1
ASL instruction



As figure 8-1 illustrates, the ASL instruction moves each bit in an 8-bit number one space to the left—each bit, that is, except bit 7. Bit 7 drops into the carry bit of the processor status register. Bit 0 of the new number, left empty by the ASL instruction, is filled with a 0.

The ASL instruction is used for many purposes in 8502 assembly language. For instance, it is often used as an easy way of multiplying numbers by 2. Listing 8-1 shows a routine that doubles numbers in a program created using a Commodore 64 Macro Assembler.

Listing 8-1
Multiplying by 2 using ASL

```
10 ;
20  *=$1300
30 ;
40  LDA #$40 ;REM 0100 0000
50  ASL A ;SHIFT VALUE IN ACCUMULATOR TO
    LEFT
60  STA $FA
70  .END
```

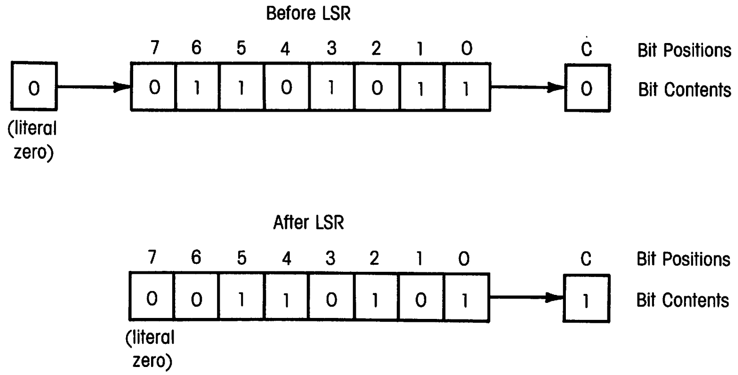
If you run the program shown in listing 8-1, and then use the C-128 monitor to examine the contents of memory address $FA, you'll see that the number $40 (0100 0000) has been doubled to $80 (1000 0000) before being stored in memory address $FA.

Another use for the ASL instruction is to "pack" data, thus increasing the computer's effective memory capacity. Later in this chapter, there is an example of how to pack data using the ASL instruction.

LSR (Logical Shift Right)

The LSR (logical shift right) instruction is the opposite of the ASL instruction, as you can see by taking a look at figure 8-2.

Figure 8-2
LSR instruction



LSR, like ASL, works on whatever binary number is in the 8502's accumulator. But LSR shifts each bit in the number one position to the right. Bit 7 of the new number, left empty by the LSR instruction, is filled with a 0. And the LSB (least significant bit), bit 0, is put into the carry flag of the P register.

The LSR instruction can be used to divide any even 8-bit number by 2, as shown in listing 8-2.

Listing 8-2
Dividing by 2 using LSR

```
10 ;
20 ;DIVIDING BY 2 USING LSR
30 ;
40 VALUE1=$FA
50 VALUE2=$FB
60 ;
70  *=$1300
80 ;
90  LDA #6 ;OR ANY OTHER 8-BIT NUMBER
100  STA VALUE1
110 ;
120 ;NOW WE'LL DIVIDE BY 2
130 ;
140  LDA VALUE1
150  LSR A
160  STA VALUE2
170  .END
```

The routine in listing 8-2 divides the number stored in VALUE1 by 2, and stores the quotient in VALUE2. As an added benefit, it can also tell you whether the number it has divided is odd or even. It leaves that bit of information (no pun intended) in the carry bit of the 8502 P register; if the routine leaves the carry bit clear, the number that was divided is odd. If the carry bit is set, the number is even.

Listing 8-3 is a program that can tell you whether a number is odd or even. The program is rather trivial, but it does illustrate an interesting point. In lines 120 and 130, a memory address labeled FLGADR (for "flag address") is cleared to 0. Then the contents of another memory location, called VALUE1, are shifted to the right one position and stored in a third address, called VALUE2. If the value being shifted is even, then the shift operation does not set the carry bit, and the subroutine ends. But if the shifting operation does set the carry bit, the program jumps to line 220, and the carry bit, now set, is rotated into FLGADR using an instruction called ROL (which you'll learn more about in a moment). So, if the routine leaves a 0 in FLGADR, the number that was divided is even. But if the routine ends with a 1 stored in FLGADR, the number that was divided is odd.

Listing 8-3
ODDTEST program

```
10 ;
20 ;ODDTEST
30 ;
40 VALUE1=$FA
50 VALUE2=$FB
60 FLGADR=$FC
70 ;
80  *=$1300
90 ;
100  LDA #7 ;(ODD)
110  STA VALUE1
```

Listing 8-3 cont.

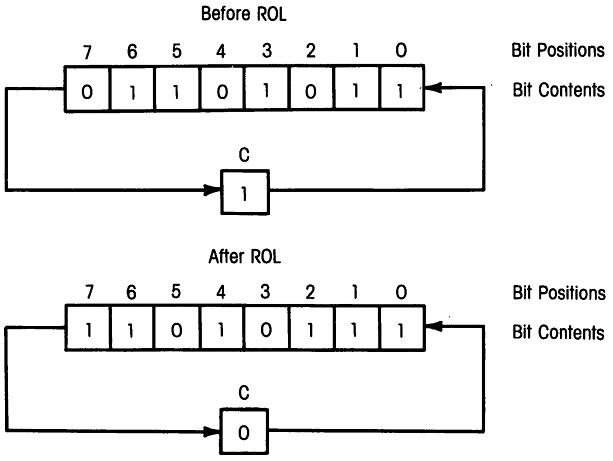
```
120  LDA #0
130  STA FLGADR ;CLEARING FLGADR
140 ;
150  LDA VALUE1
160  LSR A ;PERFORM THE DIVISION
170  STA VALUE2 ;DONE
180 ;
190  BCS FLAG
200  RTS ;END ROUTINE IF CARRY CLEAR ...
210 ;
220 FLAG
240  ROL FLGADR
250  RTS ;... AND END THE PROGRAM
```

As previously mentioned, you can also use LSR to unpack data that has been packed using ASL. But to unpack data, LSR has to be used with another assembly language instruction, called a logical operator. We'll discuss logical operators, and look at some sample routines for packing and unpacking data, later in this chapter. Meanwhile, let's examine two more bit-shifting operators: ROL (rotate left) and ROR (rotate right).

ROL (Rotate Left)

Two other instructions, ROL (rotate left) and ROR (rotate right), are also used to shift bits in binary numbers, but they use the carry bit in a different manner. They are often used in 6502/8502 multiplication and division routines, and in many other types of routines in which bits are shifted and tested. Figure 8-3 shows how the ROL instruction works.

Figure 8-3
ROL instruction

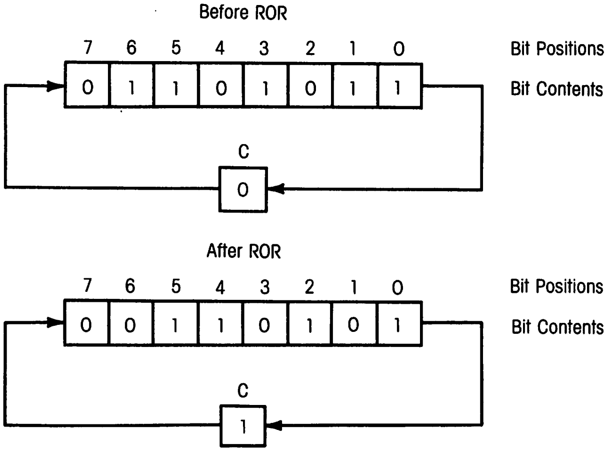


ROL, like ASL, can be used to shift the contents of the accumulator or a memory register one place to the left. But ROL does not place a 0 in the bit 0 position. Instead, it rotates the carry bit into bit 0 of the register being shifted. Then it moves every other bit in that register one place to the left, rotating bit 7 into the carry bit. If the carry bit is set when that happens, a 1 is placed in the bit 0 position of the byte being shifted. If the carry bit is clear, a 0 goes into the bit 0 position of the shifted register.

ROR (Rotate Right)

ROR works just like ROL, but in the opposite direction, as shown in figure 8-4. It moves each bit of the byte being shifted one position to the right, and rotates the carry bit into the bit 7 position of the shifted byte. As part of the same rotation process, bit 0 of the shifted byte is moved into the carry bit of the processor status register.

Figure 8-4
ROR instruction



Logical Operators

Before we move on to our next topic—binary arithmetic—let's take a brief glance at four important assembly language mnemonics called logical operators. These instructions are AND ("and"), ORA ("or"), EOR ("exclusive or"), and BIT ("bit"). (The BIT instruction is discussed at the end of this chapter.)

The four 8502 logical operators look very mysterious at first glance. But, in typical assembly language fashion, they lose much of their mystery after you understand how they work.

AND, ORA, EOR, and BIT are all used to compare values. But they work differently from the comparison operators CMP, CPX, and CPY. The instructions CMP, CPX, and CPY all yield general results; they can only determine whether two values are equal and, if the values aren't equal, which one is larger. AND, ORA, EOR, and BIT are more specific instructions. They're used to compare single bits of numbers, and hence have all sorts of uses.

The four logical operators in assembly language use the principles of a mathematical discipline called Boolean logic. In Boolean logic, the binary numbers 0 and 1 are used not to express values, but to indicate whether a statement is true or false. If a statement is proved true, its value in Boolean logic is 1. If it is false, its value is 0.

AND Operator

The operator AND in assembly language has the same meaning as the word *and* in English. If one bit AND another bit have a value of 1 (and are thus "true"), then the AND operator also yields a value of 1. But if any other condition exists—if one bit is true and the other is false, or if both bits are false—the AND operator returns a result of 0, or false.

The results of logical operators are often illustrated with diagrams called truth tables. Figure 8-5 is a truth table for the AND operator.

Figure 8-5
AND truth table

0	0	1	1
AND 0	AND 1	AND 0	AND 1
0	0	0	1

In 8502 assembly language, the AND instruction is often used in an operation called *bit masking.* The purpose of bit masking is to clear or set specific bits of a number. The AND operator can be used, for example, to clear any number of bits by placing a 0 in each bit that is to be cleared. Listing 8-4 shows how a bit-masking operation using the AND mnemonic might work in an assembly language program.

Listing 8-4
Bit masking using AND

```
100  LDA #$AA ;BINARY 1010 1010
110  AND #$F0 ;BINARY 1111 0000
```

Figure 8-6 shows the AND operation that takes place if the two lines of code in listing 8-4 are included in a program.

Figure 8-6
How AND works in bit masking

Binary Operation		Hexadecimal Equivalent
1010 1010	(contents of accumulator)	#$AA
AND 1111 0000		AND #$F0
1010 0000	(new value in accumulator)	#$A0

In the operation illustrated in figure 8-6, the low nibble of #$AA is cleared to #$00 (with a result of #$A0). This technique works with any 8-bit number. Regardless of the number passed through mask 1111 0000, the number's lower nibble is always cleared to #$00 and its upper nibble always emerges from the AND operation unchanged.

ORA Operator

When the instruction ORA ("or") compares a pair of bits, the result of the comparison is 1 (true) if the value of either bit is 1. Figure 8-7 is a truth table for the ORA instruction.

Figure 8-7
ORA truth table

0	0	1	1
ORA 0	ORA 1	ORA 0	ORA 1
0	1	1	1

ORA is also used in bit-masking operations. Listing 8-5 is an example of a masking routine using ORA.

Listing 8-5
Bit masking using ORA

```
LDA VALUE
ORA #$0F
STA DEST
```

In the segment of code illustrated in listing 8-5, suppose that the number in VALUE was $22 (binary 0010 0010). Figure 8-8 shows the masking operation that would take place.

Figure 8-8
How ORA works in bit masking

```
    0010 0010   (in accumulator)
ORA 0000 1111   (#$0F)
-------------
    0010 1111   (new value in accumulator)
```

EOR Operator

The instruction EOR (which stands for "exclusive or") returns a true value (1) if one—and only one—of the bits in the pair being tested is a 1. Figure 8-9 is a truth table for the EOR operator.

Figure 8-9
EOR truth table

0	0	1	1
EOR 0	EOR 1	EOR 0	EOR 1
0	1	1	0

The EOR instruction is often used for comparing bytes to determine if they are identical, because if any bit in the two bytes is different, the result of the EOR comparison is non-zero. Figure 8-10 shows how the EOR instruction works in a bit-masking operation.

Figure 8-10
How EOR works in bit masking

Column 1	Column 2
1011 0110	1011 0110
EOR 1011 0110	EOR 1011 0111
0000 0000	0000 0001

In column 1 of figure 8-10, the bytes being compared are identical, so the result of the comparison is zero. In column 2, one bit is different, so the result of the comparison is non-zero.

The EOR operator is also used to complement values. If an 8-bit value is EORd with $FF (1111 1111 in binary), every bit that's a 1 is complemented to a 0, and every bit that's a 0 is complemented to a 1. This process is illustrated in figure 8-11.

Figure 8-11
Complementing a number using EOR

```
    1110 0101   (in accumulator)
EOR 1111 1111
-------------
    0001 1010   (new value in accumulator)
```

Still another useful characteristic of the EOR instruction is that when EOR is performed twice on a number, using the same operand, the number is first changed to another number, and then restored to its original value. This process is illustrated in figure 8-12.

Figure 8-12
Complementing and restoring a number using EOR

Operation 1
Complementing a Number

```
    1110 0101   (in accumulator)
EOR 0101 0011
-------------
    1011 0110   (new value in accumulator)
```

Operation 2
Restoring the Original Number

```
    1011 0110   (new value in accumulator)
EOR 0101 0011   (same operand as operation 1)
-------------
    1110 0101   (original value in accumulator restored)
```

Because the EOR instruction can alter a number and then restore the number to its original value, EOR instructions are often used in graphics programs. With the help of the EOR mnemonic, a skilled programmer can erase an object from the screen, store the bytes used to display the object somewhere in memory, and then restore the object on the screen. An object can thus be made to disappear and later reappear on the screen, with the help of the EOR instruction.

Packing and Unpacking Data in Memory

Now we're ready to discuss packing and unpacking data using bit-shifting and bit-testing instructions. First, let's take a look at how you can pack data to conserve space in your computer's memory.

Packing Data

To get an idea of how data packing works, suppose that you have a series of 4-bit values stored in a block of memory. These values could be ASCII characters, BCD numbers, or just about any other kinds of 4-bit values.

Using the ASL instruction, you could pack two such values into every byte of the block of memory. Thus, you would store the values in half the memory space that they had previously occupied in their unpacked form. Listing 8-6, PACKDATA, is a routine that can be used to pack bytes of data in a program.

Listing 8-6
PACKDATA routine

```
10 ;
20 ;PACKDATA
30 ;
40  *=$1300
50 ;
60  NYB1=$FA
70  NYB2=$FB
80  PKDBYT=$FC
90 ;
100  LDA #$04 ;OR ANY OTHER 4-BIT VALUE
110  STA NYB1
120  LDA #$06 ;OR ANY OTHER 4-BIT VALUE
130  STA NYB2
140 ;
150  CLC
160  LDA NYB1
170  ASL A
180  ASL A
190  ASL A
200  ASL A
210  ORA NYB2
220  STA PKDBYT
230  RTS
```

The routine in listing 8-6 loads a 4-bit value into the accumulator, shifts that value to the high nibble in the accumulator, and then—using the ORA logical operator—places another 4-bit value in the low nibble of the accumulator. Thus, the accumulator is "packed" with two 4-bit values, and those two values are stored in PKDBYT, a single 8-bit memory address.

Type the PACKDATA program, and execute it using the assembler's machine language monitor. After you run the program, use the C-128 monitor to peek into the computer's memory to see exactly what has happened. Just type the monitor command:

```
M $FA
```

(for "display memory address $FA"), and the computer will respond with a line showing you what the program did. If all has gone well, the number $04 is stored in memory address $FA and the number $06 is stored in memory address $FB. And both of these values are packed together and stored in memory address $FC.

It doesn't take much imagination to see how this technique can increase your computer's capacity to store 4-bit numbers or ASCII characters, which can be stored in memory in the form of 4-bit numbers. By packing data, you can double the text storage capacity of an 8-bit computer, because you can store two characters in each 8-bit register in the computer's memory.

Unpacking Data

It wouldn't do much good to pack data if it couldn't later be unpacked. It so happens that data packed using ASL can be unpacked using the complementary instruction LSR (logical shift right) and the logical operator AND. Listing 8-7 is a program called UNPACKIT, which can unpack data using a series of LSR instructions.

Listing 8-7
Unpacking data using LSR

```
10 ;
20 ;UNPACKIT
30 ;
40 PKDBYT=$FA
50 LOWBYT=$FB
60 HIBYT=$FC
70 ;
80  *=$1300
90 ;
100  LDA #255 ;OR ANY OTHER 8-BIT VALUE
110  STA PKDBYT
120  LDA #0 ;CLEAR LOWBYT AND HIBYT
130  STA LOWBYT
140  STA HIBYT
150 ;
160  LDA PKDBYT
170  AND #$0F ;BINARY 0000 1111
180  STA LOWBYT
190  LDA PKDBYT
200  LSR A
210  LSR A
220  LSR A
230  LSR A
240  STA HIBYT
250  RTS
```

The UNPACKIT program works much like the PACKDATA program but in reverse. First, the accumulator is loaded with an 8-bit number into which two 4-bit values have been packed. The upper four bits of this packed byte are then filled with zeros using the logical operator AND. Then the lower nibble of the byte is stored into a memory register called LOWBYT.

Then, the accumulator is loaded for a second time with the packed byte. This time the byte is shifted four places to the right using the instruction LSR. The result of this maneuver is a 4-bit value that is finally stored in a memory register called HIBYT. The packed value in PKDBYT has thus been split, or "unpacked," into two 4-bit values—one stored in LOBYT and the other in HIBYT. And each of these 4-bit numbers (which may represent an ASCII character or any other 4-bit value) can now be processed separately.

BIT Operator

That brings us to the BIT operator, an instruction that's a little more complicated than AND, ORA, or EOR. The BIT instruction determines whether the value stored in a memory address matches a value stored in the accumulator. The BIT instruction can be used only with absolute or zero page addressing—in other words, using either of these formats:

```
BIT $02A7
BIT $FB
```

When a BIT instruction is encountered in an assembly language program, a logical AND operation is performed on the byte being tested. The value of the memory location being tested is not changed, but the *opposite* result of the AND operation is stored in the zero flag of the processor status register. In other words, if any set bits in the accumulator match any set bits stored in the same positions in the value being tested, the Z flag is cleared. If there are no set bits that match, the Z flag is set. Listing 8-8 is an assembly language routine that uses the BIT instruction.

Listing 8-8
Using the BIT instruction

```
  LDA #01
  BIT $0C00
  BNE MATCH
  JMP NOGOOD
MATCH RTS
```

In the short segment of code illustrated in listing 8-8, a check is made to determine whether bit 0 is set in the value stored in memory register $0C00. If the bit is set, the Z flag of the P register is cleared, and the program branches to the line labeled MATCH. If there is no match, the Z flag is set, and the program jumps to whatever routine has been labeled NOGOOD.

The BIT mnemonic also performs other functions. When you use the BIT instruction, bits 6 and 7 of the value being tested are always deposited directly into bits 6 and 7 of the processor status register. This can be a useful thing to know because bit 6 and bit 7 are important flags in the 8502 chip's P register; bit 6 is the P register's overflow (V) flag, and bit 7 is its negative (N) flag. Therefore, the BIT instruction can be used as a quick and easy method for checking bit 6 or bit 7 of any 8-bit value. If bit 6 of the value being tested is set, the P register's V flag will also be set, and a BVC or BVS instruction can then be used to determine what will happen next in the program. If bit 7 of the tested value is set, the P register's N flag will be set, and a BPL or BMI instruction can be used to determine the outcome of the routine.

It's also important to note that after all of these actions take place, the value in the accumulator (and the memory location being tested) always remain unchanged. So if you want to perform a logical AND operation without disturbing the value of the accumulator or the memory register being tested, the BIT mnemonic may be the best instruction to use.

9

Assembly Language Math

How the C-128 adds, subtracts, multiplies, and divides

In the first eight chapters, we've been doing a lot of reading, and a lot of writing. Now it's time to do some arithmetic. In this chapter, you'll learn how the Commodore 128 adds, subtracts, multiplies, and divides.

As we've seen in previous chapters, the Commodore 128 can deal with many kinds of numbers—including binary, decimal, hexadecimal, signed, and unsigned numbers. Other kinds of numbers that the C-128 can handle include binary coded decimal numbers and floating-point decimal numbers. We're going to take a cursory look at each of these types of numbers—and maybe a few other kinds, too.

To understand how the C-128 works with numbers, it is essential to have a fairly good understanding of the busiest flag in the 8502 microprocessor chip: the carry flag of the 8502's processor status register. So let's take a close look at the 8502's carry flag now.

A Close Look at the Carry Bit

The best way to get a closeup view of how the carry bit works is to examine it through an "electronic microscope," at the bit level. Figure 9-1 is an illustration of two short binary addition routines, neither of which generates a carry in hexadecimal or binary notation.

Figure 9-1
Addition problems without carries

Problem 1		Problem 2	
Hex	**Binary**	**Hex**	**Binary**
#$04	0100	#$08	1000
+ #$01	+ 0001	+ #$03	+ 0011
#$05	0101	#$0B	1011

Figure 9-2 presents two problems that use larger (8-bit) numbers. The first of these problems doesn't generate a carry, but the second one does.

Figure 9-2
More complex addition problems

Problem 1		Problem 2	
Hex	**Binary**	**Hex**	**Binary**
#$8E	1000 1110	#$8D	1000 1101
+ #$23	+ 0010 0011	+ #$FF	+ 1111 1111
#$B1	1011 0001	$#018C	(1) 1000 1100

Note that in the second problem in figure 9-2, the sum is a 9-bit number: 1 1000 1100 in binary, or #18C in hexadecimal notation. Listing 9-1 is an assembly language program that performs this addition problem. It is titled ADDNCARRY.

Listing 9-1
8-bit addition with a carry

```
10 ;ADDNCARRY
20  *=$1300
30  CLD
40  CLC
50  LDA #$8D
60  ADC #$FF
70  STA $FA
80  RTS
```

When you've typed and assembled the ADDNCARRY program, run it using the assembler's machine language monitor. When the program has been executed, you can use your C-128 monitor's M instruction to examine memory address $FA. If everything works as planned, $FA will hold the number #$8C. That isn't the sum of the numbers #$8D and #$FF, but it's close. In hexadecimal arithmetic, the sum of #$8D and #$FF is #$18C—exactly the sum we got, plus a carry.

So where's the carry?

Well, if everything you've read about the carry bit in these pages is true, our missing carry must be tucked away where it's supposed to be: that is, in the carry bit of the computer's processor status register. So let's go there and look for it.

Looking for a carry bit inside a Commodore 128 may seem like looking for a needle in a haystack. But a carry bit isn't too hard to find. One way to locate the carry that's missing from the ADDNCARRY program in listing 9-1 is to insert a few additional lines into the program. Listing 9-2 is an expanded version of the program, with those extra lines inserted.

Listing 9-2
Improved 8-bit addition program

```
 10 ;ADDNCARRY2
 20  *=$0C00
 30  CLD
 40  CLC
 50  LDA #$8D
 60  ADC #$FF
 70  STA $FA
 80  LDA #0
 90  ROL A
100  STA $FB
110  RTS
```

In the added lines in the ADDNCARRY2 program, the accumulator is cleared, and then the bit-shifting operator ROL rotates the P register's carry bit into the accumulator. The contents of the accumulator are then deposited into memory register $FC using an ordinary STA instruction. If this routine works, it means we've found our missing carry bit.

The best way I can think of to see if the program works is to type, assemble, and run it. After you do that, you can peek into memory addresses $FA and $FB using your C-128 monitor to see whether the calculation in the ADDNCARRY2 program resulted in a carry.

So let's do it! Assemble the program, execute it, and then use the monitor to take a look at the contents of memory address $FA. The monitor should tell you two things: that memory address $FA again holds the value #$8C—the result of our ADDNCARRY2 calculation, without its carry—and that the carry resulting from the calculation now resides in memory register $FB.

Adding 16-Bit Numbers

Now let's look at a program that adds two 16-bit numbers. The same principles used in this program can also be used to write programs that add numbers having 24 bits, 32 bits, and more. The program, called ADD16, appears in listing 9-3.

Listing 9-3
16-bit addition program

```
10 ;
20 ;ADD16
30 ;
40 ;THIS PROGRAM ADDS A 16-BIT NUMBER IN
    $FA AND $FB
50 ;TO A 16-BIT NUMBER IN $FC AND $FD
60 ;AND DEPOSITS THE RESULTS IN $0C00 AND
    $0C01
70 ;
80  *=$1300
90 ;
100  CLD
110  CLC
120  LDA $FA ;REM LOW HALF OF 16-BIT NUMBER
     IN $FA AND $FB
130  ADC $FC ;REM LOW HALF OF 16-BIT NUMBER
     IN $FC AND $FD
140  STA $0C00 ;LOW BYTE OF SUM
150  LDA $FB ;REM HIGH HALF OF 16-BIT
     NUMBER IN $FA AND $FB
160  ADC $FD ;REM HIGH HALF OF 16-BIT
     NUMBER IN $FC AND $FD
170  STA $0C01 ;HIGH BYTE OF SUM
180  RTS
```

When you examine the ADD16 program in listing 9-3, remember that the Commodore 128 stores 16-bit numbers in the reverse order from what you might expect—the low-order byte is first and the high-order byte is second. After you understand that fluke (a characteristic of all 6502/8502-based computers), 16-bit binary addition isn't hard to comprehend.

The first thing the ADD16 program does is clear the carry flag of the P register. Then the program adds the low byte of a 16-bit number in $FA and $FB to the low byte of a 16-bit number in $FC and $FD. The result of this half of our calculation is placed in memory address $0C00. If there is a carry, the P register's carry bit is set automatically.

In the second half of the program, the high byte of the number in $FA and $FB is added to the high byte of the number in $FC and $FD. If the P register's carry bit has been set as a result of the preceding addition operation, then a carry is also added to the high bytes of the two numbers. If the carry bit is clear, there is no carry.

When this half of our calculation has been completed, the result is put into memory address $0C01. Then, finally, the results of our completed addition problem are stored—low byte first—in memory addresses $0C00 and $0C01.

Subtracting 16-Bit Numbers

Because subtraction is the opposite of addition, the carry flag is set, not cleared, before a subtraction operation is performed in 8502 binary arithmetic. In subtraction, the carry flag is treated as a borrow (not a carry) and it must therefore be set (not cleared) so that if a borrow is necessary, there'll be a value to borrow from.

After the carry bit is set, an 8502 subtraction problem is quite straightforward. Listing 9-4 shows a 16-bit subtraction program. In this program, the 16-bit number in $FA and $FB is subtracted, low byte first, from the 16-bit number in $FC and $FD. The result of our subtraction problem, including a borrow from the high byte if one was necessary, is stored in memory addresses $0C00 and $0C01, in the low-byte-first convention that is typical in 6502/8502-based computers.

Listing 9-4
16-bit subtraction program

```
10 ;
20 ;SUB16
30 ;
30 ;THIS PROGRAM SUBTRACTS A 16-BIT NUMBER
    IN $FA AND $FB
40 ;FROM A 16-BIT NUMBER IN $FC AND $FD
50 ;AND DEPOSITS THE RESULTS IN $0C00 AND
    $0C00
60 ;
70  *=$1300
80 ;
90  CLD
100  SEC ;REM SET CARRY
110  LDA $FC;REM LOW HALF OF 16-BIT NUMBER
     IN $FC AND $FD
```

Listing 9-4 cont.

```
120  SBC $FA;REM LOW HALF OF 16-BIT NUMBER
     IN $FA AND $FB
130  STA $0C00 ;LOW BYTE OF THE ANSWER
140  LDA $FD ;REM HIGH HALF OF 16-BIT
     NUMBER IN $FC AND $FD
150  SBC $FB ;REM HIGH HALF OF 16-BIT
     NUMBER IN $FA AND $FB
160  STA $0C01 ;HIGH BYTE OF THE ANSWER
170  RTS
```

Multiplying Numbers

Binary numbers are multiplied in the same way as decimal numbers. Unfortunately, though, the 6502/8502 instruction set contains no specific instructions for multiplication or division. To multiply a pair of numbers using 6502/8502 assembly language, you have to perform a series of addition operations, as shown in figure 9-3. To divide numbers, you have to perform subtraction sequences.

Figure 9-3 Binary multiplication problem

```
   0110     ($06)
 X 0101     ($05)
 -------
   0110
  0000
 0110
0000
 -------
0011110     ($1E)
```

Look closely at the multiplication problem in figure 9-3, however, and you will see that it isn't difficult to split a multiplication problem into a series of addition problems. In this example, the binary number 0110 is first multiplied by 1, and the result is also 0110.

Next, 0110 is multiplied by 0. The result of that operation—a string of zeros—is shifted one space to the left. Then 0110 is multiplied by 1 again, and the result is again shifted to the left. Finally, another multiplication by 0 results in another string of zeros, which is also shifted to the left. Then, all of the partial products of our problem are added, just as they would be in a conventional multiplication problem. The result of this addition, as you can see, is $1E.

This multiplication technique works fine, but it's really quite arbitrary. Why, for example, did we shift each partial product (except the first one) in this problem to the left? We could have accomplished the same result by shifting the partial product above it to the right before adding.

In 8502 multiplication, that's what often happens; instead of shifting each partial product to the left before storing it in memory, many 8502 multiplication algorithms shift the preceding partial product to the right before adding it to the new one. Listing 9-5 demonstrates how this process works.

Listing 9-5 Multiplication program

```
10 ;
20 ;MULTA
30 ;
40 MPR=$FC ;MULTIPLIER
50 MPD1=$FD ;MULTIPLICAND
60 MPD2=$0C00 ;NEW MULTIPLICAND AFTER 8
   SHIFTS
70 PRODL=$0C01 ;LOW BYTE OF PRODUCT
80 PRODH=$0C02 ;HIGH BYTE OF PRODUCT
90 ;
100  *=$1300
110 ;
120 ;THESE ARE THE NUMBERS WE WILL MULTIPLY
130 ;
140  LDA #250
150 STA MPR
160  LDA #2
170  STA MPD1
180 ;
190 MULT CLD
200  CLC
210  LDA #0 ;CLEAR ACCUMULATOR
220  STA MPD2 ;CLEAR ADDRESS FOR SHIFTED
     MULTIPLICAND
230  STA PRODH ;CLEAR HIGH BYTE OF PRODUCT
     ADDRESS
240  STA PRODL ;CLEAR LOW BYTE OF PRODUCT
     ADDRESS
250  LDX #8 ;WE WILL USE THE X REGISTER AS
     A COUNTER
260 LOOP LSR MPR ;SHIFT MULTIPLIER RIGHT;
     LSB DROPS INTO CARRY
270  BCC NOADD ;TEST CARRY BIT; IF ZERO,
     BRANCH TO NOADD
280  LDA PRODH
290  CLC
300  ADC MPD1 ;ADD HIGH BYTE OF PRODUCT TO
     MULTIPLICAND
310  STA PRODH ;RESULT IS NEW HIGH BYTE OF
     PRODUCT
320  LDA PRODL ;LOAD ACCUMULATOR WITH LOW
     BYTE OF PRODUCT
330  ADC MPD2 ;ADD HIGH PART OF
     MULTIPLICAND
340  STA PRODL ;RESULT IS NEW LOW BYTE OF
     PRODUCT
```

Listing 9-5 cont.

```
350 NOADD ASL MPD1 ;SHIFT MULTIPLICAND
    LEFT; BIT 7 DROPS INTO CARRY
360  ROL MPD2 ;ROTATE CARRY BIT INTO BIT 7
     OF MPD1
370  DEX ;DECREMENT CONTENTS OF X REGISTER
380  BNE LOOP ;IF RESULT ISN'T ZERO, JUMP
     BACK TO LOOP
390  RTS
400  .END
```

Not a Simple Problem

As the MULTA program in listing 9-5 demonstrates, 8-bit binary multiplication isn't exactly easy. There's a lot of left and right bit shifting involved, and it's hard to keep track. In listing 9-5, the most difficult manipulation to follow is probably the one involving the multiplicand (MPD1 and MPD2). The multiplicand is only an 8-bit value, but it's treated as a 16-bit value because it keeps getting shifted to the left, and while it is moving, it takes a 16-bit address (actually two 8-bit addresses) to hold it.

To see for yourself how the program works, type and assemble it. Use the monitor's G command to execute the program. Then, while you're still in monitor mode, you can take a look at the contents of memory addresses $0C01 and $0C02. These two addresses should now hold the number $01F4 (low byte first, remember), which is the hex equivalent of the decimal number 500. And 500 is the product of the decimal number 2 and the decimal number 250, which our program multiplied.

An Improved Multiplication Program

Although the MULTA program works fine (provided you've made no mistakes typing, assembling, and running it), it isn't the only multiplication program available to C-128 assembly language programmers; in fact, it isn't even a very good one. There are many algorithms for binary multiplication, and some of them are shorter and more efficient than the one we just executed. Listing 9-6, for example, is an improved multiplication program. Titled MULTB, it's considerably shorter than the MULTA program in listing 9-5, and therefore it uses memory more efficiently and it runs faster. One of the neatest tricks in the MULTB program is that it uses the 8502's accumulator, rather than a memory address, for temporary storage of the problem's results.

Listing 9-6 Improved multiplication program

```
10 ;
20 ;MULTB
30 ;(AN IMPROVED MULTIPLICATION PROGRAM)
40 ;
50 PRODL=$FC
```

Listing 9-6 cont.

```
60 PRODH=$FD
70 MPR=$0C00
80 MPD=$0C01
90 ;
100  *=$1300
110 ;
120 VALUES LDA #10
130  STA MPR
140  LDA #10
150  STA MPD
160 ;
170  LDA #0
180  STA PRODH
190  LDX #8
200 LOOP LSR MPR
210  BCC NOADD
220  CLC
230  ADC MPD
240 NOADD ROR A
250  ROR PRODH
260  DEX
270  BNE LOOP
280  STA PRODL
290  RTS
```

If you like, you can test the MULTB multiplication program the same way you tested the previous one: by executing it using your machine language monitor, and then using your monitor to take a look at its results.

Play around with these two multiplication problems, trying out different values and seeing how these values are processed in each program. The more you experiment with binary multiplication programs, the better you'll understand them. And the better you understand binary addition, subtraction, and multiplication problems, the more understanding you'll have of 6502/8502 assembly language.

One of the best ways to become familiar with how binary multiplication works is to do a few problems by hand—using those two tools of our forefathers, a pencil and a piece of paper. Work enough binary multiplication problems on paper, and you'll soon begin to understand the principles of 6502/8502 multiplication.

Dividing Numbers

Just as subtraction is a reverse form of addition, division is a reverse form of multiplication. So it should come as no surprise that the 8502 chip, which has no specific instructions for multiplying numbers, also lacks specific instructions for dividing.

Still, it is possible to perform division—even multiprecision long division—using instructions that are available to the 8502 microprocessor. As we have seen, the 8502 chip can multiply numbers, provided that the multiplication problems are broken down into sequences of addition problems. Likewise, the 8502 chip can divide numbers if the division problems are broken down into sequences of subtraction problems. Listing 9-7 is a program that divides one number into another number by breaking the division process down into a series of subtraction routines.

Listing 9-7
Binary long division program

```
10 ;
20 ;DIV8/16
30 ;
40  *=$1300
50 ;
60 DVDH=$FC ;HIGH PART OF DIVIDEND
70 DVDL=$FD ;LOW PART OF DIVIDEND
80 QUOT=$0C00 ;QUOTIENT
90 DIVS=$0C01 ;DIVISOR
100 RMDR=$0C02 ;REMAINDER
110 ;
120  LDA #$1C ;JUST A SAMPLE VALUE
130  STA DVDL
140  LDA #$02 ;THE DIVIDEND IS NOW $021C
150  STA DVDH
160  LDA #$05 ;ANOTHER SAMPLE VALUE
170  STA DIVS ;WE'RE DIVIDING BY 5
180  ;
190  LDA DVDH ;ACCUMULATOR WILL HOLD DVDH
200  LDX #08 ;FOR AN 8-BIT DIVISOR
210  SEC
220  SBC DIVS
230 DLOOP PHP ;SAVE P REGISTER (ROL & ASL
     AFFECT IT)
240  ROL QUOT
250  ASL DVDL
260  ROL A
270  PLP ;RESTORE P REGISTER
280  BCC ADDIT
290  SBC DIVS
300  JMP NEXT
310 ADDIT ADC DIVS
320 NEXT DEX
330  BNE DLOOP
340  BCS FINI
350  ADC DIVS
360  CLC
370 FINI ROL QUOT
380  STA RMDR
390  RTS
```

During the execution of the program, titled DIV8/16, the high byte of the dividend is stored in the accumulator, and the low byte of the dividend is stored in a variable called DVDL. The program does a lot of looping, shifting, rotating, and subtracting. When everything is finished, the quotient is stored in a variable labeled QUOT, and the quotient's remainder is stored in the accumulator. Next, in line 380, the remainder is moved out of the accumulator and into a variable called RMDR. Then, finally, an RTS instruction ends the program.

The DIV8/16 program can be used to divide any unsigned 16-bit number by any unsigned 8-bit number. As written, it divides the hexadecimal number $021C (540 in decimal notation) by 5. The quotient is stored in memory address $0C00, and the remainder, if any, is stored in memory register $0C02.

Type, assemble, and run the program, and then use your monitor to inspect the contents of memory addresses $0C00 and $0C02. Address $0C00 should now hold the hexadecimal number $6C (108 in decimal notation), and address $0C02 should hold a 0, because the quotient of 540 divided by 5 is 108, with no remainder.

As you will discover as you work through the DIV8/16 program, it's even more difficult to write a C-128 division routine than it is to write a C-128 multiplication program. In fact, writing just about any kind of multiprecision math program for an 8-bit computer is usually more trouble than it's worth. When you write a program that has to make just a few calculations, sometimes you can use short, simple routines such as the ones presented in this chapter. But assembly language is usually not the best language to use for writing long, complex programs that contain a lot of multiprecision math. If you ever have to write such a program, you might find it worthwhile to write part of the program in assembly language and the other part—the part with the mathematics—in BASIC. That way, you can take advantage of the excellent floating-point math package that's built into the BASIC 7.0 interpreter in the Commodore 128. If you can't do that, it might still be best to write the program in some language—almost any language—besides assembly language. Because of the extraordinary amount of work that it takes to write mathematical routines for computers in the Commodore class, it's usually much better to write complex mathematical programs in BASIC, Pascal, COBOL, Logo, or almost any other high-level programming language than it is to try to write them in assembly language.

If, despite this warning, you still have a yen to write complex math routines in 6502/8502 assembly language, there are a few books that may provide you with some help. There are quite a few type-and-run math routines in some of the manuals and texts listed in the Bibliography. One text that contains an abundance of fairly complex math routines that are yours for the typing is *6502 Assembly Language Subroutines,* written by Lance A. Leventhal and Winthrop Saville and published by Osborne/McGraw Hill.

Signed Numbers

Before we move to the next chapter, it might be a good idea to pause for a moment and take a quick look at signed numbers. To represent a signed number in binary arithmetic, the leftmost bit (bit 7) represents a positive or negative sign. In signed binary arithmetic, if bit 7 of a number is a 0, the number is positive. But if bit 7 is a 1, the number is negative.

If you use one bit of an 8-bit number to represent its sign, you no longer have an 8-bit number. What you then have is a 7-bit number—or, expressed another way, you have a signed number that can represent values from −128 to +127, instead of 0 to 255.

It takes more than redesignating a bit to turn unsigned binary arithmetical operations into signed binary arithmetical operations. Consider, for example, what would happen if we tried to add the numbers +5 and −4 by doing nothing more than using bit 7 as a sign, as shown in figure 9-4.

Figure 9-4 An attempt to add two signed numbers

```
  0000 0101 (+5)
+ 1000 0100 (−4)
----------------
  1000 1001 (−9)
```

That answer is wrong. The answer should be +1. The reason we arrived at the wrong answer is that we tried to solve the problem without using a concept fundamental to the use of signed binary arithmetic: the concept of complements.

Complements are used in signed binary arithmetic because negative numbers are complements of positive numbers. Complements of numbers are very easy to calculate in binary arithmetic; in binary math, the complement of a 0 is a 1, and the complement of a 1 is a 0.

Ones Complement Addition

It might be reasonable to assume that the negative complement of a positive binary number could be arrived at by complementing each 0 in the number to a 1, and each 1 to a 0 (except for bit 7, which must be used for representing the number's sign). The technique of calculating the complement of a number by flipping its bits from 0 to 1 and from 1 to 0 has a name in assembly language circles. It's called *ones complement.*

To see if the ones complement technique works, let's try using it to add two signed numbers: +8 and −5, as shown in figure 9-5.

Figure 9-5 Another attempt to add two signed numbers

```
  0000 1000   (+8)
+ 1111 1010   (−5) (ones complement)
-----------
  0000 0010   (+2) (plus carry)
```

Oops! That's wrong, too! The answer should be +3. Well, that takes us back to the drawing board. Ones complement arithmetic doesn't work.

Twos Complement Addition

But there's another technique—which comes very close to ones complement—that does work. It's called *twos complement,* and it works as follows. First calculate the ones complement of a positive number. Then simply add one. That gives you the twos complement—the true complement—of the number.

Then you can use the conventional rules of binary math on signed numbers and, if you don't make any mistakes, they'll work every time. Figure 9-6 shows this procedure in action.

Figure 9-6
Twos complement addition

```
  0000 0101  (+5)
+ 1111 1000  (−8) (twos complement)
  ---------
  1111 1101  (−3)
```

Figure 9-7 shows another twos complement addition problem.

Figure 9-7
Another example of twos complement addition

```
  1111 1011  (−5) (twos complement)
+ 0000 1000  (+8)
  ---------
  0000 0011  (+3) (plus carry)
```

Although it isn't easy to understand why twos complement arithmetic works, here are some facts that may help. Because the highest bit of a binary number is always interpreted as a sign in twos complement notation, a binary number that has its highest bit set is always interpreted as a negative number. So the hexadecimal number $7F, which equates to the decimal number 127, is the highest positive number that can be expressed in 8-bit twos complement notation. Increment the hex number $7F, and you'll see why this is true. In binary notation, $7F is written %0111 1111—a binary number in which the high bit is not set. But if you increment $7F, you'll get $80, or %1000 0000—a number that has its high bit set, and will therefore be interpreted in 8-bit twos complement notation as −128, not +128. So the largest positive number that can be expressed in 8-bit twos complement notation is 127.

Now let's take a look at some negative binary numbers. In twos complement arithmetic, negative numbers start at −1 and work backwards, just as conventional negative numbers do in ordinary arithmetic. In conventional arithmetic, there's no such number as −0, so when you decrement a 0, what you get is not −0, but −1. Decrement −1, and what you get is −2, which may look at first glance like a larger number, but is really a smaller one. Decrement −2, and you get −3. And so on.

Twos complement arithmetic works in a similar fashion. Decrement 0 using 8-bit twos complement arithmetic, and you get not −0 (because there's no such number as −0) but $FF, which equates to −1 in decimal notation. Decrement $FF in twos complement, and you get $FE, the 8-bit signed binary equivalent of −2. The decimal number −3 is written $FD in 8-bit twos complement notation, and the decimal number −4 is written $FC. And so on.

Keep working backwards, and you'll eventually discover that the smallest negative number that can be expressed in 8-bit twos complement notation is the hexadecimal number $7F, which equates to −127 in decimal.

So twos complement arithmetic works every time—almost. To make the process completely accurate, we have to consider one more factor: the state of the overflow (V) flag of the 8502 processor status register.

The 8502 overflow flag, which we discussed briefly in chapter 3, is used as a carry flag in arithmetical operations that involve signed numbers. The reason is not difficult to understand. As we have seen, bit 7 of a signed 8-bit binary number is not actually a part of the number; it is merely the number's sign. So the highest bit in an 8-bit signed number is bit 6, not bit 7.

Unfortunately, though, twos complement arithmetic operates so automatically and transparently that the 8502 chip never really knows whether it's working with signed or unsigned numbers. So when it performs addition or subtraction operations on large numbers, there is sometimes an overflow from bit 6 to bit 7, and this overflow sometimes causes an unwanted change in the sign of the result. The overflow flag can prevent this kind of mathematical disaster—but before you can use the overflow flag, you have to know how. So here's how the overflow flag works.

Before an addition operation is performed on a pair of signed numbers, you should clear the overflow flag. In an addition operation that involves signed numbers, an overflow condition occurs when the numbers being added have the same sign, but their sum has a different sign. If this condition occurs during a signed number addition operation, the overflow flag is set, and the sign that has been altered by "accident" can be changed back to its desired state. It is the responsibility of the programmer, however, to read the overflow flag and then make the necessary sign change.

The overflow flag should also be set prior to a subtraction operation involving a pair of signed numbers. In a signed number subtraction operation, an overflow condition occurs if bit 7 of the minuend and bit 7 of the subtrahend are different, and bit 7 of the result is the value of the subtrahend. The P register's overflow flag is also set when this kind of condition occurs, and again it is the responsibility of the programmer to take corrective action.

10

Memory Magic

Managing the C-128's memory

The engineers who designed the Commodore 128 accomplished quite a feat; they stuffed 196K of memory into a 64K computer, and then they made the machine expandable to 388K. Just how they did it is the topic of this chapter. By the time we move to chapter 11, you'll know how to read a C-128 memory map—and where you can and can't store programs in your computer's memory.

From a programmer's point of view, as well as from a user's point of view, the C-128 is three computers in one. It can be used as a Commodore 64 computer, as a fully equipped 80-column CP/M computer, or in its own C-128 mode. In each of its three modes, the C-128 manages its memory in a different manner. When the computer is in C-64 mode, it works exactly like a Commodore 64. When used as a CP/M computer, it utilizes a memory layout specifically designed to work with the CP/M operating system. When the computer is operated in its own C-128 mode, the number of memory configurations that can be used with it is virtually unlimited.

The secret behind the C-128's versatility is a most unusual internal architecture. As we have seen in previous chapters, the C-128 is equipped with two central processors: an 8510 chip, which is downwardly compatible with the 6510 chip used in the Commodore 64, and a Z-80 microprocessor, which can be used to write, as well as run, programs designed for computers equipped with the CP/M operating system.

Because the C-128 is three computers in one, it has three separate memory maps, and each one is completely different. Let's take a look now at each of these three memory maps, beginning with the one that the C-128 uses when it's running CP/M programs.

Commodore 128's CP/M Mode

When the Commodore 128 is operated in CP/M mode, it uses both of its built-in central processor units, or CPUs. Because CP/M programs are written in Z-80 assembly language, the C-128 uses a Z-80 chip as its primary processor when it is running in CP/M mode. But even then, the computer uses its 8502 chip to handle disk operations, telecommunications operations, and other kinds of input/output functions. So when the C-128 is being used as a CP/M computer, it has to switch between its Z-80 chip and its 8502 chip, handling programs and data with one chip and various kinds of I/O operations with the other.

In addition to its built-in Z-80 chip, the C-128 also contains a block of ROM specially designed to support CP/M operations. When the computer is operated in CP/M mode, it switches out all ROM designed for 8502 operations, and switches in its block of specially engineered CP/M ROM.

The C-128's block of CP/M ROM is physically situated at memory locations $D000 through $DFFF, but the C-128's Z-80 chip "sees" it at memory addresses $00 through $0FFF. By fooling the Z-80 in this fashion, the C-128 manages to keep its page zero and stack addresses free for use by its 8502 chip, simultaneously providing its Z-80 chip with a page zero and a stack of its own.

A memory map of the C-128 when it is operated in C/PM mode is shown in figure F-3 (appendix F). In this mode, the C-128 uses two blocks of memory, each with a capacity of 64K. On the memory map in figure F-3, these blocks of memory are labeled bank 0 and bank 1.

The largest single block of memory on the CP/M map is a 56K segment of RAM called the *transient program area,* or *TPA.* This block of memory, which occupies most of memory bank 1, is where CP/M programs and operating system commands are stored in the C-128's memory. Bank 1 also contains the CP/M page zero, plus two blocks of memory that it shares with bank 0: a small *memory management unit,* or *MMU,* and a memory segment called the *BDOS* and *BIOS* common area.

The MMU, which is also used when the C-128's 8502 chip is active, resides at the very top of the computer's memory. We'll take a closer look at it later in this chapter. BDOS, which stands for *basic disk operating system,* is a CP/M utility that handles file-based disk operations and also has limited screen editing and printing capabilities. BIOS, which stands for *basic input/output system,* is a CP/M utility that manages simple input and output operations.

Memory bank 0 also contains the C-128's CP/M ROM, which extends from $00100 to $0FFF (as far as the Z-80 chip is concerned); some special-purpose RAM blocks used for key code tables and screen displays; a buffer in which a CCP (command interpreter) is stored; and the BDOS and BIOS memory area that is shared with bank 1.

As we have seen in previous chapters, the main reason that CP/M capabilities were included in the C-128 was to give C-128 users access to the tremendous amount of CP/M business software. It is possible to write CP/M programs on a Commodore 128, and Commodore even offers a CP/M programming kit (available at extra cost) to users who are interested in writing C-128 programs in Z-80 assembly language. But the C-128 was not designed primarily as a CP/M software development system and, because there isn't too much CP/M software being developed these days, it is unlikely that many C-128 owners have much interest in doing Z-80 assembly language programming. So, instead of spending any more time discussing CP/M memory, let's move on to the C-128's other two operating modes, which will probably be of much greater interest to most C-128 users.

Commodore 128's C-64 Mode

When the Commodore 128 is operated in C-64 mode, it doesn't just operate like a Commodore 64; from a functional point of view, it *is* a Commodore 64, albeit in Commodore 128 clothing.

The Commodore 64 that's built into the Commodore 128 has a BASIC interpreter and a kernel operating system that are identical to those built into a stand-alone C-64. Its ROM and I/O functions, just like those in a standard C-64, can be switched out to give the user access to more RAM. All of its I/O features work like those of a standard Commodore 64. So, when the C-128 is operated in its C-64 mode, it is virtually one hundred percent compatible with software designed for the Commodore 64.

Not Exactly Alike

There are a few differences, however, between the original C-64 and the twin C-64 that's built into the C-128. One difference is that the C-64 housed inside the C-128 does not have a 6510 chip of its own, but uses the C-128's 8502 chip. Another difference is that the original C-64 was equipped with a video chip called the 6567 VIC-II, and the C-64 that's built into the C-128 uses a newer and more advanced video chip—the same chip that the C-128 uses to generate its own 40-column display. This new chip, called the 8564 VIC-II, can do everything that the original VIC-II can do—and a little more. It contains circuitry that increases the operating speed of the C-128 and enables the C-128 to read the twenty-four new keys that the C-64 does not have.

Theoretically, all of these enhancements could be used to improve software written for the C-128's built-in C-64. But from a practical standpoint, it is unlikely that much use will be made of them in software developed for the C-64 from now on. Reason: The Commodore 128 is so superior to the Commodore 64, with so many improvements and added features, that it doesn't make much sense for software developers to write programs for the C-64 that's built into the C-128 when they could be writing real C-128 programs.

Because the C-128 and the C-64 are so closely related, though, we will examine the memory maps of both machines in some detail. The Commodore 64 and the Commodore 128 are alike in many respects, and the areas in which they are alike are pointed out as we examine the memory map of the C-128's built-in C-64. So now let's take a closeup look at the memory map of the Commodore 64.

C-64 Memory Map

The Commodore 128's C-64 memory map is illustrated in appendix F, figure F-4. From the standpoint of memory organization, the Commodore 64 was—and still is—a rare breed of computer. Most 64K computers have about 48K of addressable RAM, plus about 16K of ROM, for a total of 64K. But the Commodore 64 has a full 64K of user-addressable RAM, plus 24K of built-in ROM. When you add its 64K of RAM and 24K of ROM, you get 88K of built-in memory.

This 88K of memory is controlled by a pair of special memory registers that occupy memory addresses $0000 and $0001. These two special registers are also active when the Commodore 128 is in its native C-128 mode. However, because their original design was based on the memory configuration of the Commodore 64, they are used more frequently in Commodore 64 programming than they are in programs designed for the Commodore 128.

In the following section, we will see how memory registers $0000 and $0001 are used in Commodore 64 assembly language programs. Later, we will see how they are used in Commodore 128 programs.

R6510 and D6510 I/O Port Registers

In C-128 and C-64 literature, register $0000 is referred to as the *6510/8502 I/O port data direction register,* and register $0001 is called the *6510/8502 I/O port data register.* These two registers were all that distinguished the 6510 chip used in the original Commodore 64 from its predecessor, the original 6502. When the 8502 chip was designed, registers $0000 and $0001 were retained.

Memory register $0000, the 8502 I/O port data direction register, is often labeled D6510 in assembly language programs. The D6510 register is used to control the direction of data flow into and out of certain blocks of memory, and also the direction of data flow to and from the Commodore 64/128 I/O data register.

Memory register $0001 is called the 8502 chip's I/O port data register. In assembly language programs, it is often labeled R6510. In C-64 and C-128 programs, the chief function of the R6510 register is to determine which blocks of memory will be used as RAM and which blocks will be used as ROM.

The R6510 and D6510 have eight bits each, but only six of the bits in each register are significant. There is a one-to-one correspondence between these six bits in each register; register $0001 (R6510) controls the flow of data into and out of the C-128, and register $0000 (D6510) controls the direction in which the data flows.

Bits 6 and 7 of the D6510 and R6510 registers are not significant. Bits 3 through 5 of each register control a data cassette recorder if the computer is connected to a cassette machine. Bits 0, 1, and 2 of each register determine whether specific blocks of the Commodore 128's memory will be used as ROM or RAM, and what kinds of data will appear in the blocks being used as RAM.

When you turn on the Commodore 128 or the Commodore 64, the six significant bits of the D6510 register are set as shown in figure 10-1. There is rarely any need to change these bit settings in a user-written C-128 or C-64 program. The bits in the Commodore 64/128's other "magic register"—register R6510—are reset more often.

The R6510 register, like the D6510, has eight bits, five of which

Figure 10-1
Significant bits of
the D6510 register

5	4	3	2	1	0	Bit Positions
1	0	1	1	1	1	Bit Contents

are significant. When an R6510 bit is set, the function that it controls becomes an output function. When a bit is cleared, the function that it controls becomes an input function.

Bits 6 and 7 of the R6510 register, like the corresponding bits of the D6510 register, are not significant. Bits 3 through 5, like the same bits of the D6510 register, are used to control the Commodore 64/128 data cassette recorder. That leaves only three bits—bits 0 through 2—for memory control purposes, but these are three of the most powerful bits on the Commodore's memory map.

Table 10-1 shows the six significant bits of the R6510 and D6510 registers, along with their functions.

Table 10-1
Functions of the R6510 and D6510 Registers

Bit	Name	Setting at Power Up	Function
0	LORAM	1 (Output)	On: $A000-$BFFF is BASIC ROM Off: $A000-$BFFF is RAM
1	HIRAM	1 (Output)	On: $E000-$FFFF is kernel ROM Off: $E000-$FFFF is RAM
2	CHAREN	1 (Output)	On: $D000-$DFFF is I/O ROM Off: $D000-$DFFF is character ROM
3		1 (Output)	On: Write to cassette line Off: Read from cassette line
4		0 (Input)	On: Cassette switch pressed Off: Cassette switch not pressed
5		1 (Output)	On: Cassette motor on

Bit 0: LORAM—As table 10-1 illustrates, bit 0 of the R6510 register (the LORAM bit) controls whether memory addresses $A000 through $BFFF will be used as BASIC ROM or as user-addressable RAM. If bit 0 is set, the Commodore 64/128's built-in BASIC interpreter uses memory registers $A000 through $BFFF, and the interpreter can be used for writing and running BASIC programs. If bit 0 of the R6510 register is cleared, memory addresses $A000 through $BFFF can be used as free RAM, and the computer's built-in BASIC interpreter is not available for use in writing or running Commodore 64/128 BASIC programs.

Bit 1: HIRAM—Bit 1 of the R6510 register (the HIRAM bit) controls whether the Commodore 64/128 kernel will occupy memory registers $E000 through $FFFF, or whether those registers will be available for use as free RAM.

Bit 2: CHAREN—Bit 2 of the R6510 register (the CHAREN bit) is the "magic bit" that determines whether memory addresses $D000 through $DFFF will be used as RAM registers by the Commodore 64/128's operating system or as character generator ROM. When bit 2 of the R6510 register is set, the $D000 through $DFFF block of memory is used as RAM by the computer's operating system, primarily the portion of the operating system that controls the operation of I/O devices. When the CHAREN bit is clear, all RAM stored in the $D000 through $DFFF area becomes temporarily inaccessible, and 4K of character generator ROM, better known as the Commodore 64/128's built-in character set, is switched in.

In the Commodore 128, the CHAREN bit cannot be directly accessed from user-written programs. In the C-128, the CHAREN bit is accessed through a "shadow register" at memory address $D9. When the value 4 is stored in $D9, the C-128's operating system sets the CHAREN bit, and the VIC-II then "sees" I/O ROM when it looks at memory addresses $D000 through $DFFF. When a zero is stored in $D9, the CHAREN bit is cleared by the operating system, and the VIC chip sees character generator data in the block of memory at $D000 through $DFFF.

Map Reading

In the Commodore 64, the D6510 and R6510 registers work hand in hand with four preset memory configurations called *memory banks.* The C-128 also uses these four memory banks, though in a slightly different manner. We'll see how all of this works in the second half of this chapter.

First, though, here's a simplified explanation of the Commodore 128/64 memory map. It covers most of the important memory blocks in the computer's four main memory banks.

$0000-$00FF: Page Zero RAM—In both the Commodore 64 and the Commodore 128, the block of memory that extends from $0000 to $00FF is known as page zero. Page zero appears in this segment of memory in all 16 of the C-128's memory banks, so you don't need to use any bank-switching techniques to access page zero in a C-128 program.

Page zero is a very important block of memory because data stored there can be accessed using a 1-byte operand—a procedure that saves both memory and processing time. In addition, two addressing modes—indirect indexed addressing and indexed indirect addressing—will not work unless their operands have zero page addresses.

Because zero page addresses offer important benefits, page zero is a desirable district on the memory map of the Commodore 128. The engineers who designed the C-128 claimed most of it for themselves to set up the computer's operating system and BASIC interpreter. Consequently, very little space on page zero is available for use by user-written C-128 programs.

When you're writing a program for the Commodore 128, as we'll see a little later in this chapter, there is a way to increase the amount of space available on page zero by using some fancy bank-switching techniques. But if you don't want to go through the trouble of doing that, it's absolutely essential to find at least a few free memory locations on page zero. Unless you take special steps to increase the number of memory locations, that's how many free memory locations you'll find on page zero: a few. Table 10-2 shows some of the C-128's most important page zero memory locations.

Table 10-2
C-128's Page Zero Memory Map

Memory Addresses	Register's Function
$00-$01	Special 8502 I/O control registers (the I/O port data direction register and the I/O port data register)
$02-$D6	Used by BASIC and the C-128 operating system
$D7	Screen width flag (0 = 40 columns, 128 = 80 columns)
$D8	40-column text/graphics mode flag (224 is graphic 4, 160 is graphic 3, 96 is graphic 2, 32 is graphic 1, 0 is graphic 0)
$D9	Shadow register for CHAREN bit of memory address $0001 (the 8502 I/O port data register). 4 is I/O ROM at $D000-$DFFF, 0 is character ROM at $D000-$DFFF.
$E0-$F9	Flags used in windowing, screen editing, and keyboard reading operations
$FA-$FE	Bytes left free for user-written programs
$FF	Used by BASIC interpreter

As table 10-2 illustrates, there are only five bytes on page zero that have been left free for use in user-written programs—$FA, $FB, $FC, $FD, and $FE. Despite this scarcity of page zero space, however, a sharp-eyed programmer can usually find other zero page addresses that can be safely used in assembly language programs. For example, many of the addresses on page zero are reserved for use by the C-128's built-in BASIC 7.0 interpreter, and a number of others are only used by the floating-point math routines built into the computer's operating system. Many of the registers in these categories can be used by user-written programs, if the programs aren't designed to be called from BASIC and don't require the C-128's floating-point math package.

Detailed C-128 memory maps can be found in a number of reference volumes, including the *Commodore 128 Reference Guide for Programmers,* written by David L. Heiserman and published by Howard W. Sams & Co., Inc.

$0100-$01FF: 8502 Stack—The RAM space that extends from $0100 through $01FF in the Commodore 128 is reserved for use by the 8502 stack. The stack appears in this segment of RAM in all 16 of the C-128's memory banks, so you don't have to do any bank-switching operations to access the stack. However, as we shall see later in the chapter, bank-switching operations can be used to move the C-128 stack, thus permitting you to assign individual stacks to individual programs.

The stack, which was introduced in chapter 6, is a section of memory that the Commodore 128 uses to keep track of the return addresses of machine language subroutines and interrupts (temporary interruptions in normal program processing). The stack is also used for temporary storage of the values of memory registers during operations that would otherwise change those values and thus destroy them. The stack is frequently used by the C-128 operating system, and is also available for use in user-written programs.

$0200-$03FF: Kernel RAM and Free RAM—The block of memory that extends from $0200 through $03FF contains important RAM vectors and routines, and is shared by all 16 of the computer's memory banks. So this block of memory is always accessible to user-written programs, no matter which memory bank is active.

$0400-$07FF: Video Memory RAM (Text Mode) and Color RAM (Bit-Mapped Mode)—In all 16 memory banks of the Commodore 128, the block of memory that extends from $0400 through $07FF is ordinarily used as a screen map for 40-column text—that is, for the storage of data that generates 40-column text displays. Other blocks of memory can be used for the same purpose, but the block of memory at addresses $0400 through $07FF is the C-64/C-128's default screen map; it is the RAM block used as a screen map when you first turn on the C-64 or the C-128.

The $0400 through $07FF memory block is not used as a screen map, though, when the C-128 is in its high-resolution, or bit-mapped, graphics mode. When you use high-resolution graphics, this segment of memory is too small to hold a complete screen map. A 40-column screen map uses only 1,000 bytes of memory, but a high-resolution screen map requires 8,000 bytes. So, when you operate the C-128 in high-resolution graphics mode, a larger block of memory must be designated as a screen map, and the $0400 through $07FF memory block can then be used to control the colors that appear on the screen. These procedures are examined in greater detail in part 2, which is devoted to C-128 graphics and sound.

$0800-$1BFF (Bank 0): BASIC and Kernel Working Storage—The memory space that extends from $0800 to $1BFF in bank 0 is reserved for use by BASIC and kernel routines. However, several blocks of RAM in this area are often available for use in user-written programs. They are:

- Addresses $0B00 through $0BFF. This 255-byte block of RAM acts as a buffer when a data cassette recorder is being used. But it can be used as free RAM in programs that do not make use of a cassette recorder.
- Addresses $0C00 through $0DFF. This 512-byte block of RAM (which immediately follows block $0B00 through $0BFF in the C-128's memory) is normally set aside for use as a text buffer in programs that use the C-128's RS-232

serial port. But in programs that do not make use of the serial port, it is available as free RAM.

- Addresses $0E00 through $0FFF. This 512-byte memory segment of RAM (which comes after the $0C00 through $0DFF block) is used for storing sprite definitions. In programs that don't use sprites, it can be used as free RAM.
- Addresses $1300 through $1BFF. This block of memory can provide more than 2K of free RAM to programs that do not use foreign language keys or the C-128's function keys. This segment of memory is used by many of the assembly language programs in this volume.

$1C00-$FEFF (Bank 0): BASIC Program Text Storage—The text of programs written in BASIC is stored in the block of memory that extends from $1C00 to $FEFF in memory bank 0. In programs that do not make use of BASIC, this block of memory can be used either as a high-resolution screen map or as free RAM.

In programs that use both BASIC and high-resolution graphics, you can use the BASIC statement GRAPHIC 1 to move the BASIC text storage area up to memory address $4000. After you issue a GRAPHIC 1 command, the $1C00 through $FEFF memory block can be used as a high-resolution screen map, and the area of memory above $4000 can be used as a BASIC program text storage area. Alternatively, you can issue the GRAPHIC 0 command immediately after the GRAPHIC 1 command; the C-128 will return to text mode, but the start of the BASIC text storage area is *not* moved back to $1C00.

There are other ways to change the location of the BASIC text storage area. For example, a new address can be stored, low byte first, in BASIC's bottom-of-memory pointer, which is at memory addresses $2D and $2C (45 and 46 in decimal notation). This method has one drawback; it can't be used in a BASIC program because it would destroy the program. However, BASIC's top-of-memory pointer, at memory addresses $39 and $40 (57 and 58 in decimal notation), can be changed from BASIC. You can lower the top of BASIC by storing a new ending address (low byte first) in $39 and $40; then the space above that address is free for use by machine language programs.

$1C00-$3FFF (Bank 0): High-Resolution Screen Data and Color Memory (Bit-Mapped Mode)—When the Commodore 128 is in its 40-column high-resolution graphics mode, the block of RAM that extends from $1C00 to $1FFF is used as color memory and the $2000 through $3FFF memory block holds high-resolution screen data. When high-resolution graphics are not in use, this block of memory can be used either for the storage of BASIC program text or as free RAM.

$0800-$FEFF (Bank 1): BASIC Variable Storage—The block of memory that extends from $0800 to $FEFF in bank 1 is reserved for use as a storage area for BASIC 7.0 variables. In C-128 programs that do not make use of BASIC, this block of memory is available for use as free RAM.

$4000-$FF00 (Bank 15): BASIC and Kernel ROM—On the C-128 memory map, bank 15 extends from memory address $4000 through memory address $FF00 and is made up completely of ROM, arranged as follows:

$4000-$AFFF	BASIC ROM
$B000-$BFFF	Monitor ROM
$C000	Screen editor ROM
$D000-$DFFF	I/O or character generator ROM (depending on the contents of memory address $D9)
$E000-$FFEF	Kernel ROM

$FF00-$FF04: C-128 MMU—The memory management unit (MMU) register, which controls the memory resources of the Commodore 128, occupies memory addresses $FF00 through $FF04 in all 16 of the computer's memory banks.

Character Sets

The Commodore 64 and the Commodore 128 each have two built-in character sets. One character set, which contains uppercase letters and graphics characters, begins at memory address $D000 in both the Commodore 64 and the Commodore 128. The other set includes uppercase and lowercase letters but no graphics characters. It starts at $D800 in both machines. Complete listings of both character sets can be found in appendix C and appendix D.

In the C-64 and C-128, either or both character sets can be copied from ROM into RAM. After a character set is copied into RAM, it can be modified in any way that the programmer desires. Copying a character set into RAM, and then modifying it, is the main topic for discussion for the rest of this chapter.

Because the memory architecture of Commodore 128 is based on the memory architecture of the Commodore 64, understanding how memory transfer operations work in the C-64 will help us understand how the same kinds of operations work in the C-128. So now let's take another look at the Commodore 64 memory map.

Four-Bank C-64

The memory map of the C-64 (shown in figure F-5 in appendix F) can be divided into four memory banks. The reason for this arrangement is that the VIC-II chip, which generates the 40-column screen display of both the C-128 and its built-in C-64, can access only 16K of memory at a time. By dividing the C-64's 64K of memory into four banks of 16K each, the engineers who designed the C-64 gave the VIC-II chip a choice of four memory banks to look at. By setting certain bits in certain memory registers, you can tell the VIC-II chip which bank of memory to access, and which starting address in that memory to look at, to get the data needed to produce screen displays.

Selecting a Memory Bank

To tell the VIC-II chip which C-64 memory bank to access, a code number must be stored in a special memory register known as CI2PRA. In both the Commodore 64 and the Commodore 128, the CI2PRA register occupies memory address $DD00.

Table 10-3 shows what values must be stored in bits 0 and 1 of memory register $DD00 to determine which memory bank will be used for the storage of graphics data.

Table 10-3
Selecting a C-64 Memory Bank

To Select Bank:	At These Addresses:	Store These Values in Bits 0 and 1 of Memory Register $DD00:	
		Binary	Decimal/Hexadecimal
0	$0000-$3FFF	00	00
1	$4000-$7FFF	01	01
2	$8000-$BFFF	10	02
3	$C000-$FFFF	11	03

An Important Warning

There's one important note of caution regarding the use of table 10-3. Memory register $DD00 is a multipurpose register that controls various input/output functions of the 8502 chip and designates the memory bank that will be used by the VIC-II chip. So, when you load a value from table 10-3 into bits 0 and 1 of register $DD00, don't disturb the contents of the other bits in the register. To alter bits 0 and 1 without disturbing bits 2 through 7, you can use an assembly language routine such as the one shown in listing 10-1.

Listing 10-1
Altering $DD00 using a masking operation

```
LDA $DD00
AND #$FC ;CLEAR BITS 0 AND 1
ORA #$02 ;A VALUE FROM TABLE 10-3
STA $DD00
```

Selecting a Starting Address for a Character Set

After you have designated a memory bank for a character set, you can tell the VIC-II chip where to look within that memory bank to find your character set. To do this, store a code number into three bits of another special register: the VIC-II memory control register, which occupies memory address $D018 and is often labeled VMCSB. To tell the C-64's VIC-II chip where a copied character set resides in

RAM, store the proper code in bits 0 through 3 of the VMCSB register, as illustrated in table 10-4.

Table 10-4 Starting Addresses of C-64 Character Set in RAM

Store starting address code in $D018 (VMCSB) as follows:

			Starting Addresses		
Bits to Set	**Hex Number**	**Bank 0**	**Bank 1**	**Bank 2**	**Bank 3**
XXXX111X	$0E	$3800	$7800	$B800	N/A*
XXXX110X	$0C	$3000	$7000	$B000	N/A*
XXXX101X	$0A	$2800	$6800	$A800	N/A*
XXXX100X	$08	$2000	$6000	$A000	N/A*
XXXX011X	$06	$1800†	$5800	$9800†	N/A*
XXXX010X	$04	$1000†	$5000	$9000†	N/A*
XXXX001X	$02	$0800	$4800	$8800	$C800
XXXX000X	$00	N/A*	$4000	$8000	$C000

*This memory block is not normally available for storage of character data.
†This block is where ROM character images are stored; RAM stored in this block is not visible to the VIC-II chip, and thus cannot be used for storage of user-generated character sets or any other kinds of graphics data.

As you can see by examining table 10-4, any character set copied into RAM—whether it's a full set or a partial set—must begin at a memory address that's evenly divisible by $800—that is, on a 2K boundary. Table 10-4 shows every possible starting address for a C-64 character set. As we shall see later, the Commodore 128 uses the same set of character set starting addresses, but handles them a little differently.

As table 10-4 also shows, the meaning of the value stored in bits 0 through 3 of the C-64's VMCSB register can vary, depending upon which bank of memory the VIC-II chip is looking at.

Another Important Warning

Here's another word of caution: Bits 1 through 3 of the C-64's VMCSB register are used to indicate the starting address of character sets, and bits 4 through 7 of the same register are used to inform the VIC-II chip of the starting address of screen memory. So when you use bits 1 through 3 of this register, you have to be careful not to disturb any values stored in bits 4 through 7. (You don't have to worry about bit 0 because it isn't significant.)

To set bits 1 through 3 of the VMCSB ($D018) register without disturbing the upper four bits, you can use a masking routine like the one shown in listing 10-2. To find out what value to use in the third line of this listing, merely consult table 10-4.

Listing 10-2 Altering VMCSB's lower nibble

```
LDA VMCSB
AND #$F0 ;CLEAR LOWER NIBBLE
ORA #$0E ;(SAMPLE VALUE FROM TABLE 10-4)
STA VMCSB
```

Now we're ready see how a character set can be copied from RAM into ROM in the Commodore 64. Then we'll see how a very similar process can be used to move a Commodore 128 character set from ROM into RAM.

Moving the C-64's Screen Map

To generate a screen display, the VIC-II chip must be told not only where to find the data it needs to produce a character set, but also where to find the bit map data that it needs to generate a screen display.

The address of the C-64's screen map, as previously noted, is determined by the contents of the upper four bits of the VMCSB register (memory register $D018). Table 10-5 shows the values that can be stored in memory register $D018 to designate the starting address of a screen display.

Table 10-5
Starting Addresses of Screen Memory in C-64

Store starting address code in $D018 (VMCSB) as follows:

	Hex		Starting Addresses		
Bits to Set	**Number**	**Bank 0**	**Bank 1**	**Bank 2**	**Bank 3**
1111XXXX	$F0	$3C00	$7C00	$BC00	N/A*
1110XXXX	$E0	$3800	$7800	$B800	N/A*
1101XXXX	$D0	$3400	$7400	$B400	N/A*
1100XXXX	$C0	$3000	$7000	$B000	N/A*
1011XXXX	$B0	$2C00	$6C00	$AC00	N/A*
1010XXXX	$A0	$2800	$6800	$A800	N/A*
1001XXXX	$90	$2400	$6400	$A400	N/A*
1000XXXX	$80	$2000	$6000	$A000	N/A*
0111XXXX	$70	$1C00†	$5C00	$9C00†	N/A*
0110XXXX	$60	$1800†	$5800	$9800†	$D800‡
0101XXXX	$50	$1400†	$5400	$9400†	N/A*
0100XXXX	$40	$1000†	$5000	$9000†	N/A*
0011XXXX	$30	$0C00	$4C00	$8C00	$CC00
0010XXXX	$20	$0800	$4800	$8800	$C800
0001XXXX	$10	$0400	$4400	$8400	$C400
0000XXXX	$00	N/A*	$4000	$8000	$C000

*This memory block is not normally available for storage of screen memory data.
†This block are where ROM character images are stored; RAM stored in this block is not visible to the VIC-II chip, and thus cannot be used for storage of other kinds of graphics data.
‡This is the default storage area for color memory. This memory block is not large enough for storage of a high-resolution screen map, and must be used to store color data when the C-64 is in text mode. So this address is normally not available for storage of screen map data in either text mode or bit-mapped mode.

When the values shown in table 10-5 are stored in the VMCSB register, care must be taken not to disturb the value of the lower nibble of the register because that nibble controls the location of the C-64 character set. A masking routine like the one shown in listing 10-3 can be used to change the upper nibble of VMCSB without changing the register's lower nibble.

Listing 10-3
Altering VMCSB's
upper nibble

```
LDA VMCSB
AND #$0F ;CLEAR UPPER NIBBLE
ORA #$80 ;(SAMPLE VALUE FROM TABLE 10-5)
STA VMCSB
```

Commodore 128's Native Mode

Now we're ready to see how the Commodore 128 works when it is operated in its C-128 mode. The Commodore 128 derives its name from the fact that it comes equipped with 128K of RAM. But calling the Commodore 128 a Commodore 128 is an understatement. Actually, the C-128 comes with a whopping 196K of memory: 128K of RAM and 68K of ROM. This prodigious amount of memory is laid out as follows:

- 128K of RAM (divided into two blocks of 64K each)
- 32K of BASIC 7.0 ROM
- An 8K operating system, which includes a user-addressable kernel
- 4K of screen editor ROM
- 4K of character data ROM
- 4K of CP/M ROM
- 16K of 80-column screen RAM (built into the C-128's 8563 VDC chip, which handles the computer's 80-column screen display)

Add all of that RAM and ROM, and you get a grand total of 196K. And that's not all that the Commodore 128 offers. In addition to the two 64K RAM banks that are built into every C-128, two more 64K RAM banks can be installed, for a total of 256K of RAM. It's also possible to add up to 64K of additional ROM—32K in the form of plug-in cartridges, and 32K that can be installed on the main circuit board. So the C-128, which has 196K of memory as soon as you take it out of the box, can be expanded (theoretically, anyway) into a 388K machine!

Those are impressive figures, especially when you consider that the 8502 chip used in the C-128, like the 6510 chip built into the C-64, is an 8-bit microprocessor. As we saw in chapter 3, an 8-bit microprocessor can address only 64K of memory at a time. So, even though the C-128 can store large amounts of data in its memory, it can't manipulate all of that data simultaneously.

Bank Switching

To handle the vast number of bytes that it can store, the C-128 relies on a rather sophisticated programming technique called *bank switching.* This technique (which was also used to expand the Apple IIc and the Apple IIe into 128K 8-bit computers) is illustrated in figures F-1 and F-2 (in appendix F).

When the Commodore 128 is operated in C-128 mode, its memory can be divided into three blocks: two 64K blocks of RAM and one 48K memory block made up almost completely of ROM. On the screen map shown in figure F-1 these segments are labeled block A, block B, and block C. (Technically, the C-128 also has two additional RAM blocks, but both of these blocks were designed for future expansion; in an unexpanded C-128, they are identical to RAM blocks A and B.)

As figure F-1 also shows, RAM block A, RAM block B, and ROM block C all share a small strip of RAM at the very top of the C-128's memory. This segment of memory is the memory management unit (MMU). It's only five bytes long—it extends from memory address $FF00 to address $FF04—but it manages all of the C-128's bank-switching operations. Because the MMU can be accessed from any block of memory, it can be used as a main switching station, moving from one memory block to another as it keeps watch over all of them simultaneously.

Actually, the configuration register at $FF00 is a RAM copy of an I/O register at $D500. Functionally, these two registers are identical; writing a value to either one of them automatically writes the same value to the other. But using the register at $FF00 is usually better than using the one at $D500, because $FF00 is accessible to all memory banks and $D500 is not. So $FF00 is referred to as the MMU configuration register throughout the rest of this chapter.

At the bottom of the screen map in figure F-1, there's another segment of RAM that's shared by blocks A and B. This portion of memory, which extends from $0000 to $03FF, is occupied by page zero ($0000-$00FF), the 8510 stack ($0100-$01FF), and a 512K block of RAM vectors, important BASIC routines, and operating system routines ($0200-$03FF). The $0000 through $03FF block of memory contains RAM, so it isn't accessible to the C-128's ROM block. But its contents are always available to RAM blocks A and B.

Three-Bank BASIC

An interesting fact about the C-128 is that its built-in BASIC 7.0 interpreter makes use of all three of the memory blocks illustrated in figure F-1. Although the C-128's BASIC interpreter resides in the ROM block, the RAM in which BASIC 7.0 programs are stored is in block A, and the variables used in BASIC 7.0 programs are stored in the 64K of free RAM that's available in block B. So, when the C-128 is running a BASIC 7.0 program, the computer's MMU is almost constantly busy switching between one block of memory and another. All of this MMU activity is usually quite transparent to the BASIC programmer because the C-128 is designed to take care of BASIC's bank-switching needs automatically.

When the Commodore 128 is processing an assembly language program, however, there is nothing automatic about bank switching.

When an assembly language program is written for the C-128, it's the programmer's responsibility to take care of all necessary bank-switching operations.

Fortunately, though, with the help of memory maps such as those shown in figures F-1 and F-2, the concept of bank switching isn't too difficult to understand. Because the 8502 chip can "see" only 64K of memory at a time, it is up to the MMU to determine whether the 8502 is looking at block A, block B, or the ROM block at any given time. To help it carry out this task, the MMU is equipped with one very special register called a configuration register, which is situated at memory address $FF00. The configuration register has eight bits, which function as follows:

- Bit 0 determines whether addresses $D000 through $DFFF in the ROM block contain I/O ROM or character data. If bit 0 of the configuration register is clear, then addresses $D000 through $D7FF and $DC00 through $DFFF contain I/O ROM, and addresses $D800 through $DBFF contain color RAM for the C-128's 40-column screen. If bit 0 is set, then addresses $D000 through $DFFF contain character generator data. Bit 0 is significant only if the ROM block is being accessed. When RAM block A or RAM block B is being accessed, addresses $D000 through $DFFF contain RAM.
- Bit 1 of the configuration register determines whether the 8502 accesses BASIC ROM or external function ROM (a ROM cartridge) when it looks at addresses $4000 through $7FFF in the ROM block. This bit is also significant only when the ROM block is being accessed. When RAM block A or RAM block B is being accessed, addresses $4000 through $7FFF contain RAM.
- Bits 2 and 3 determine whether the 8502 will see BASIC ROM, external ROM (a cartridge), or RAM when it looks at addresses $8000 through $BFFF. The settings of these bits are as follows:

00	BASIC ROM
01	Internal function ROM (not currently used)
10	External function ROM (cartridge)
11	RAM

- Bits 4 and 5 determine whether the 8502 will see BASIC ROM, external ROM (a cartridge), or RAM when it looks at addresses $C000 through $CFFF and $E000 through $FEFF. The settings of these bits are as follows:

00	BASIC ROM
01	Internal function ROM (not currently used)
10	External function ROM (cartridge)

11	RAM

- Bits 6 and 7 determine whether the 8502 will see RAM from block A or RAM from block B in memory addresses $0000 through $FFEF. The settings of these bits are:

00	RAM from block A
01	RAM from block B
10	RAM from block A
11	RAM from block B

Using the C-128's Memory Banks

As you can imagine, figuring out what memory blocks to use, and how to use them, can be quite a challenging feat for the Commodore 128 assembly language programmer. Fortunately, the engineers who designed the C-128 have provided us with a number of programming aids to make the task a little easier. For example, the C-128 has 16 preset memory configurations that can be incorporated into any program in any order with the help of an easy-to-use kernel call. Each of these configurations is called a memory bank—a term that can be somewhat confusing because the word *bank,* in this context, refers to a preset memory configuration rather than a contiguous block of memory.

To lessen the confusion a little, it is helpful to know that most of the Commodore 128's 16 memory banks will rarely, if ever, be of much concern to the average C-128 user. Some of the banks are identical to others, and a number of them are designed to be used with memory expansion cartridges and other kinds of ROMs. When we eliminate the memory banks that are not often (or never) used, only four important memory configurations remain. These banks and their contents are illustrated in table 10-6 and figure F-2. A complete list of all C-128 memory banks, and the blocks of memory that each bank contain, can be found in chapter 14 of the *Commodore 128 Reference Guide for Programmers,* written by David L. Heiserman and published by Howard W. Sams & Co., Inc.

As you can see by examining table 10-6, memory banks 0 and 1 are very similar, and banks 14 and 15 are also similar. From $0000 through $3FFF, in fact, all four banks are identical; they all contain RAM from block A. From $4000 through $FFEF, banks 0 and 1 both contain RAM, but bank 0 takes its RAM from block A and bank 1 takes its RAM from block B.

Banks 14 and 15 are identical except for the segment of memory that extends from $D000 through $DFFF. In this range of memory, bank 14 contains character ROM, and bank 15 contains I/O ROM and 40-column color RAM. In all four banks, $FF00 through $FF04 are occupied by the MMU. In banks 0 and 1, though, the MMU is surrounded by RAM, and in banks 14 and 15 it is encircled by ROM.

Table 10-6
The Four Most Important Memory Banks Used by the C-128

Bank Number	Addresses	Contents
0	$0000-$FEFF	RAM from block A
	$FF00-$FF04	MMU
	$FF05-$FFFF	RAM from block A
1	$0000-$03FF	RAM from block A
	$0400-$FEFF	RAM from block B
	$FF00-$FF04	MMU
	$FF05-$FFFF	RAM from block A
14	$0000-$3FFF	RAM from block A
	$4000-$BFFF	BASIC ROM
	$C000-$CFFF	80-column screen editor ROM
	$D000-$DFFF	Character ROM
	$E000-$FEFF	Kernel ROM
	$FF00-$FF04	MMU
	$FF05-$FFFF	Kernel ROM
15	$0000-$3FFF	RAM from block A
	$4000-$BFFF	BASIC ROM
	$C000-$CFFF	80-column screen editor ROM
	$D000-$DFFF	I/O and 40-column color map
	$E000-$FEFF	Kernel ROM
	$FF00-$FF04	MMU
	$FF05-$FFFF	Kernel ROM

Bank Switching Using BASIC

When a C-128 user is programming in BASIC, there's a convenient BANK instruction that can be used to switch from one memory bank to another. To move from one block to another in BASIC, just follow the BANK instruction with the number of the bank you want to switch to—BANK 0 to switch to bank 0, BANK 1 to switch to bank 1, and so on.

Bank Switching Using MMU Registers

Things are not that simple in C-128 assembly language, the most direct way to place a value in memory address $FF00 (the 8510 configuration register). As pointed out earlier in this section, however, there is little apparent similarity between the bank numbers used by the C-128 and the number that must be stored in $FF00 to switch from one bank to another. Table 10-7 lists the numbers of the 16 memory banks in the C-128, and the value that must be stored in the C-128 configuration register to access each bank.

Fortunately, at least one of the MMU access codes listed in table 10-7 is easy to remember; to switch to memory bank 15, the bank used to access most of the C-128's ROM, just store a 0 in the MMU's configuration register.

Another way to switch to bank 0, bank 1, or bank 14 (this trick works only with these three memory banks) is to use three MMU registers called *preconfiguration registers.* These three registers have

Table 10-7 Bank Numbers and MMU Access Codes Compared

Bank Number	MMU Access Code
0	$3F
1	$7F
2	$BF
3	$FF
4	$16
5	$56
6	$96
7	$D6
8	$2A
9	$6A
10	$AA
11	$EA
12	$06
13	$0A
14	$01
15	$00

the following addresses: $FF01, $FF02, and $FF03. To use one of these preconfiguration registers, just store a value—any value—in the register you want to use. Storing a value in $FF01 will switch to bank 0, placing a value in $FF02 will switch to bank 1, and placing a value in $FF03 will switch to bank 14.

When you use an MMU preconfiguration register, it is important to remember that it is the act of writing to the register, not the value stored in the register, that causes a bank-switching operation to take place. The C-128 refreshes all three preconfiguration registers 60 times every second. So if you write a value to one of the registers and then read it, the register will probably contain a default value rather than the value you placed in it. The default values of the C-128's preconfiguration registers are: $3F for $FF01, $7F for $FF02, and $01 for $FF03.

Bank Switching Using the C-128 Kernel

The C-128 kernel also offers some handy techniques for switching from one memory bank to another. One of these is a kernel vector called GETCFG, which can be accessed by a JSR to memory address $FF6B. To use the GETCFG subroutine, load the 8502's X register with the actual number of the bank you want to switch to, and then jump to memory address $FF6B using a JSR instruction. The value that must be stored in $FF00 to switch to the desired bank is then returned in the accumulator, and a switch can be made to that bank with the assembly language statement STA $FF00.

To use a GETCFG call, you must be in bank 15 because that's the "home" bank of the kernel ROM where the GETCFG subroutine resides. This restriction also applies to all other kernel-based bank-switching utilities. Two other kernel subroutines that you can use in bank-switching operations are INDFET, which has a jump address of $FF74, and INDSTA, which has a jump address of $FF77. INDFET can fetch a byte from any bank, without leaving the bank currently being accessed. INDSTA is a subroutine that does the opposite; it can store a byte in any bank without leaving the bank currently being accessed.

Both INDFET and INDSTA use a special kind of addressing technique that works much like indirect indexed (zero page,Y) addressing. Indirect indexed addressing, as you may recall from chapter 6, is the indirect addressing mode that is written using (*nn*),Y (where *nn* is a zero page pointer to a memory address.

To use the INDFET kernel call, first store the base address of the byte you want to access in a zero page pointer. (This is the same procedure you followed when using standard indirect indexed addressing.) Next, load the accumulator with the pointer's zero page address, and load the Y register with an offset (or with a zero if no offset is used). Then call INDFET with the assembly language statement JSR $FF74. If all goes well, INDFET returns with the desired byte stored in the accumulator, but the C-128 remains in the memory bank that it started out in.

INDSTA works much like INDFET, but in the opposite direction. To use INDSTA, store the base address of the desired byte in a zero page pointer, store the address of the pointer in memory address $02B9, load the accumulator with the byte to be stored, load the X register with the bank number, load the Y register with the index, and do a JSR to $FF77. The desired byte will be stored at the desired address in the desired bank, but the original bank setting of the C-128 will not change.

Copying a Character Set into RAM

Now we're ready to see how a character set can be copied from ROM into RAM in a Commodore 128 program. Relocating a character set in a C-128 program is much like moving a character set in a C-64 program. In fact, it's easier to list the differences between a C-64 and a C-128 character-copying operation than it is to list the similarities. So here's a summary of those differences:

1. Before a character set can be copied from ROM to RAM in a Commodore 128 program, a memory bank for the new character set must be selected by using the 8502's MMU (memory management unit).
2. In a character-copying routine written for the C-128, you cannot directly access the CHAREN bit of the I/O port data register to determine whether the VIC-II chip "sees" I/O data or character data at memory addresses $D000 through $DFFF. Instead, the CHAREN bit must be accessed by a "shadow register" at memory address $D9. A zero must be stored in $D9 so that the VIC-II chip will be able to access character data.
3. In a C-128 character-moving routine, the VMCSB register at memory address $D018, which tells the VIC-II chip where to

look for character and screen data, cannot be accessed directly. Instead, the C-128 accesses the VMCSB register through a shadow register that resides at memory address $A2C. So, instead of storing code numbers in the high nibble and the low nibble of the VMCSB register as shown in tables 10-4 and 10-5, a C-128 program must store the same numbers in memory address $A2C.

The three programs at the end of this chapter—listings 10-4, 10-5, and 10-6—demonstrate how the C-128's built-in character set can be copied into RAM, modified, and then used in its modified form in user-written programs.

Listing 10-4, called COPYCHRS.BAS, is a BASIC program that copies the C-128 character set from bank 15 ROM into bank 0 RAM. The program then modifies the @ character into a hollow square, and uses that square as a cursor in another program that allows the user to type text on the Commodore 128 screen.

Listing 10-4
COPYCHRS.BAS
program

```
5 REM **** COPYCHRS.BAS ****
7 REM
8 REM A PROGRAM TO MOVE THE C-128'S
  CHARACTER SET FROM ROM INTO RAM
9 REM
10 DATA 255,129,129,129,129,129,129,255
15 REM
30 GRAPHIC 2,1:REM MOVE START OF BASIC UP
   TO $4000
40 POKE 2604,PEEK(2604) AND 240 OR 8:REM
   TELL VIC CHIP WHERE TO FIND NEW CHAR SET
50 FAST:REM SPEED UP CHAR-COPYING OPERATION
60 FOR L=0 TO 2047:BANK 14:C=PEEK(53248+L)
   :BANK 0:POKE 8192+L,C:NEXT L:REM POKE
   CHAR DATA INTO NEW LOCATION
70 SLOW:REM RESUME NORMAL CPU SPEED
80 COLOR 0,7:COLOR 4,7:COLOR 5,2:REM SET
   SCREEN, BORDER AND CHAR COLORS
90 FOR L=0 TO 7:READ S:POKE
   8192+0*8+L,S:NEXT L:REM CHANGE '@' CHAR
   TO A BOX
100 GRAPHIC 0,1:REM USE 40-COL TEXT MODE
110 PRINT "@";:REM USE REDEFINED '@' CHAR
    AS A CURSOR
120 GETKEY A$:PRINT CHR$(20);:PRINT A$;:REM
    GET INPUT, BACKSPACE TO COVER UP
    CURSOR, AND PRINT TYPED CHAR ON SCREEN
130 GOTO 110:REM GET NEXT INPUT CHAR
```

One important part of the COPYCHRS.BAS program is in line 30. In that line, the start of memory used by BASIC is moved up to memory address $4000 so that the new character set being created by the program will not interfere with the program that is creating it.

In lines 20 and 40, the VIC-II chip is told the location of the new character set. In lines 50 through 70, the C-128's ROM character set is copied from bank 15 ROM into bank 0 RAM. The new character set starts at RAM address $2000—a block of memory reserved for a bit-mapped screen when high-resolution graphics are used, but free for just about any other kind of use when BASIC is moved out of the way and high-resolution graphics are not needed.

Type and run the COPYCHRS.BAS program, and you'll see that it takes quite a long time to copy a character set using BASIC, even when the speed of operation of the 8502 chip is increased with a FAST instruction. Listing 10-5, titled COPYCHRS2.BAS, improves matters considerably by breaking out of BASIC for a while and calling a machine language routine.

Listing 10-5
COPYCHRS2.BAS program

```
5 REM **** COPYCHRS2.BAS ****
7 REM
8 REM A PROGRAM TO MOVE THE C-128'S
  CHARACTER SET FROM ROM INTO RAM
9 REM
10 DATA 255,129,129,129,129,129,129,255
15 REM
30 GRAPHIC 2,1:REM MOVE START OF BASIC UP
   TO $4000
40 POKE 2604,PEEK(2604) AND 240 OR 8:REM
   TELL VIC CHIP WHERE TO FIND NEW CHAR SET
60 BLOAD "COPYCHRS.OBJ":SYS 4864
80 COLOR 0,7:COLOR 4,7:COLOR 5,2:REM SET
   SCREEN, BORDER AND CHAR COLORS
90 FOR L=0 TO 7:READ S:POKE
   8192+0*8+L,S:NEXT L:REM CHANGE '@' CHAR
   TO A BOX
100 GRAPHIC 0,1:REM USE 40-COL TEXT MODE
110 PRINT "@";:REM USE REDEFINED '@' CHAR
    AS A CURSOR
120 GETKEY A$:PRINT CHR$(20);:PRINT A$;:REM
    GET INPUT, BACKSPACE TO COVER UP
    CURSOR, AND PRINT TYPED CHAR ON SCREEN
130 GOTO 110:REM GET NEXT INPUT CHAR
```

COPYCHRS2.BAS is just like COPYCHRS.BAS, except lines 50 through 70 are removed and replaced by a line that loads and executes a machine language routine called COPYCHRS.O. COPYCHRS.O is generated by a source code program called COPYCHRS.S, which appears in listing 10-6. COPYCHRS.S was written using a TSDS assembler, but with minor modifications it can be typed and assembled using any assembler compatible with the Commodore 128 or the Commodore 64.

Listing 10-6 COPYCHRS.S program

```
1000 ;
1010 ; COPYCHRS.S
1020 ;
1030 *=$1300
1040 ;
1050 CHRBAS = $D000 ;START OF CHR ROM
1060 NEWADR = $2000 ;START OF NEW CHR RAM
1070 TABLEN = $800 ;LENGTH OF CHR ROM
1080 MVSRCE = $FA ;PTR TO $D000
1090 MVDEST = MVSRCE+2 ;PTR TO $2000
1100 LENPTR = $C3 ;TEMP ADR FOR TABLEN
1110 GETCFG = $FF6B ;KERNEL BANK-SWITCHING
     SUBROUTINE
1120 INDFET = $FF74
1130 ;
1140 ; POKE CHR DATA INTO NEW LOCATION
1150 ;
1160  LDA #<CHRBAS
1170  STA MVSRCE
1180  LDA #>CHRBAS
1190  STA MVSRCE+1
1200 ;
1210  LDA #<NEWADR
1220  STA MVDEST
1230  LDA #>NEWADR
1240  STA MVDEST+1
1250 ;
1260  LDA #<TABLEN
1270  STA LENPTR
1280  LDA #>TABLEN
1290  STA LENPTR+1
1300 ;
1310 ; MOVE STARTS HERE
1320 ;
1330  LDA #0
1340  STA $FF00 ;USE BANK 15
1350  LDY #0
1360  LDX LENPTR+1
1370  BEQ MVPART
1380 MVPAGE JSR GETDATA
1390  INY
1400  BNE MVPAGE
1410  INC MVSRCE+1
1420  INC MVDEST+1
1430  DEX
1440  BNE MVPAGE
1450 MVPART LDX LENPTR
1460  BEQ MVEXIT
1470 MVLAST JSR GETDATA
```

Listing 10-6 cont.

```
1480  INY
1490  DEX
1500  BNE MVLAST
1510 MVEXIT LDA #0
1520  STA $FF00 ;USE BANK 15
1530  RTS
1540 ;
1550 ; SUBROUTINE TO STORE (MVSRCE),Y IN
     (MVDEST),Y
1560 ;
1570 GETDATA PHA
1580  TXA
1590  PHA
1600  LDA #MVSRCE
1610  LDX #14 ;GET DATA FROM BANK 14
1620  JSR INDFET
1630  JSR STORDATA ;IN BANK 0
1640  PLA
1650  TAX
1660  PLA
1670  RTS
1680 ;
1690 STORDATA
1700  STA $FF01 ;USE BANK 0
1710  STA (MVDEST),Y
1720  LDA #0 ;RETURN TO BANK 15
1730  STA $FF00
1740  RTS
```

As you can see by checking out line 1340 of the COPYCHRS.S program in listing 10-6, the program does most of its work while in memory bank 15, the home bank of the C-128's BASIC 7.0 interpreter. To move the C-128's character set from ROM into bank 0 RAM, the program uses a standard kind of memory-copying algorithm that extends from line 1140 to line 1540. Line 1030 stores the program in a block of memory starting at memory address $1300 in bank 0. (This block of memory is usually available for use by assembly language programs that are short to medium in length because it is only used by foreign language utilities and function key definitions, and it contains more than 2K of RAM.)

To fetch character data from bank 14, the COPYCHRS.S program uses a subroutine called GETDATA, which starts at line 1570. GETDATA uses the kernel call INDFET to fetch the data it needs, and then uses a subroutine called STORDATA (which starts at line 1690) to store the data in bank 0 RAM. STORDATA places an arbitrary value in MMU register $FF01 to switch to bank 0, and then uses a standard indirect indexed instruction—STA (MVDEST),Y—to store the data in bank 0. Then it returns to bank 15 by storing a 0 in $FF00.

Assemble the COPYCHRS.S program and store it on a disk, and then run it using the COPYCHRS2.BAS program. Then you'll see how an assembly language routine can speed up procedures that involve extensive use of the CPU, such as character-copying operations.

Moving Page Zero and the Stack

Before we continue to chapter 11, there is one more topic that should be brought up at this point: a bank-switching procedure that allows both page zero and the 8502 stack to be moved in a C-128 program.

As we have seen in this chapter and previous chapters, page zero and the 8502 stack are very important segments of memory in assembly language programs. Free space on page zero is very limited, and it is sometimes desirable to create user-defined stacks for use in subroutines. Because of these factors and others (for example, to make it possible to have several page zeros available for use in the same program), the engineers who designed the C-128 created a method for setting up a system that enables you to move page zero, or the stack, or both, to any memory location (or pair of memory locations) in bank 0 on the C-128's memory map.

The system that carries out these operations is known as *virtual paging*. As we saw earlier in this chapter, page zero occupies memory addresses $00 through $FF in the C-128's memory, and the 8502 stack occupies memory addresses $100 through $1FF. When virtual paging is used to move page zero or the stack, neither block of memory actually goes anywhere, but the C-128's operating system is "fooled" into thinking that they are somewhere else—namely, in the location that they have been "moved" to. Any new stack or page zero location must start on a page boundary—that is, at an address divisible by $100—for the new stack or page zero to work properly.

To relocate page zero using this virtual paging system, store the high-order byte of the new location of your new page zero in memory address $D057. To relocate the stack, you can store the high byte of your new stack address in memory address $D059.

This whole process is quite simple, but three important notes of caution about relocating page zero are in order. First, when you relocate page zero, the addresses of the I/O port data direction register ($0000) and the I/O port data register ($0001) do not change. These important memory registers remain where they are, and writing to their new page zero locations has no effect. So, if you have changed the location of page zero and then want to address either of these registers ($0000 or $0001), you should use the register's original zero page address, not its apparent new one.

Another word of warning is this: After you have relocated page zero, any value that you write to page zero is actually stored in the block of memory you have designated as your new page. For example, if page zero is relocated to $1300 and a program contains the statement STA $C0, the value of the accumulator is stored not in the zero page address $C0 but in the new zero page address $13C0.

The third thing to remember when you have relocated page zero is that when you write to an address in the block of memory to which page zero has been moved, you're actually writing to the zero page address that corresponds to the address you're writing to. For example, if page zero has been relocated to $1300 and the statement STA $13C0 appears in a program, the value of the accumulator is stored not in memory address $13C0 but in the zero page address $C0.

It is also important to remember that before the stack is relocated for use in a subroutine, the stack pointer should be saved somewhere in memory before the new stack is used. Then, when the subroutine that uses its own stack has been executed, the C-128's original stack can be restored.

This brings us to the close of chapter 10. In chapter 11, we'll see how to make a character-copying routine run even faster by converting the routine into a one hundred percent machine language program.

Part Two

Assembly Language Graphics and Sound

11

Introducing Commodore 128 High-Resolution Graphics

And joystick operations

In chapter 10, we were briefly introduced to Commodore 128 graphics. In this chapter, we'll discover more of the C-128's wonderful graphics capabililties. Then we'll be ready to start writing some real razzle-dazzle graphics programs.

You'll get a chance to type, assemble, and run a program that can turn your Commodore 128's screen into a bit-mapped graphics tablet. The program is called SKETCHER, and when you understand how it works, you'll be able to create high-resolution pictures on your C-128 screen using a light pen, trackball, or joystick controller. In later chapters, you'll learn how to create custom designed characters, how to create giant headline characters, how to program and animate sprites, and how to create music and special effects on the Commodore 128.

C-128 Screen Memory

We'll start with a BASIC program titled BALLBOUNCE.BAS, which is shown in listing 11-1. It uses the Commodore 128's 40-column text mode rather than its high-resolution mode, but it creates a colorful display with some entertaining animation—and it also illustrates some important principles about Commodore 128 graphics. So please don't skip over this little program; some of the principles in this program will be encountered in the next few chapters, when we start examining more complex graphics programs written in assembly language.

Listing 11-1 BALLBOUNCE.BAS program

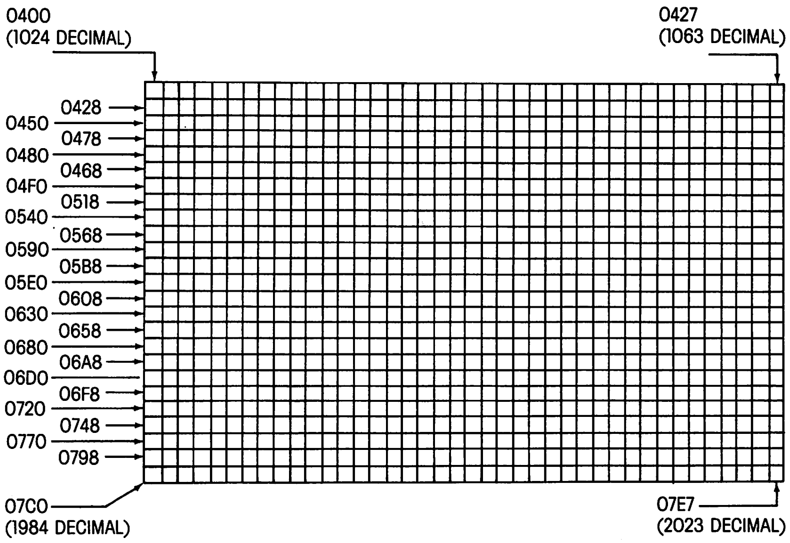
```
10 REM **** BALLBOUNCE.BAS ****
20 PRINT CHR$(147):REM CLEAR SCREEN
30 BALL=81:SPACE=96:RULE=99:REM CODES TO
   PRINT THINGS ON THE SCREEN
40 FOR L=55616 TO 55975:POKE L,2:NEXT L:REM
   MAKE BALL RED
50 FOR L=55976 TO 56015:POKE L,7:NEXT L:REM
   MAKE FLOOR YELLOW
60 POKE 53281,0:POKE 53280,6:REM BLACK
   BACKGROUND, BLUE BORDER
70 PRINT CHR$(5):REM WHITE TEXT
80 PRINT:PRINT:PRINT "    FOLLOW THE
   BOUNCING BALL . . ."
90 FOR L=1704 TO 1743:POKE L,RULE:NEXT
   L:REM DRAW FLOOR
100 PSN=1664:CT=1:REM STARTING POSITION AND
    FRAME COUNTER
110 FOR INC=1 TO 8:GOSUB 210:REM THIS LOOP
    DRAWS THE BALL GOING UP
120 PSN=PSN-40+1:REM THE BALL GOES UP
130 IF CT>40 THEN PSN=1344:CT=1:GOTO 150:
    REM BALL OFF SCREEN--BACK TO BEGINNING
140 NEXT INC
```

Listing 11-1 cont.

```
150 FOR DEC=1 TO 8:GOSUB 210:REM THIS LOOP
    DRAWS THE BALL COMING DOWN
160 PSN=PSN+41:REM THE BALL COMES DOWN
170 IF CT>40 THEN 100:REM BALL OFF SCREEN--
    LOOP BACK
180 NEXT DEC
190 GOTO 110:REM DONE--START AGAIN
200 REM **** PRINT BALL ON SCREEN ****
210 POKE PSN,BALL
220 FOR L=1 TO 50:NEXT L
230 POKE PSN,SPACE
240 CT=CT+1:RETURN
250 END
```

If you're an old hand at Commodore programming, you may know that the C-128 can display up to 1,000 characters at a time on its screen, in a format that measures 40 columns by 25 lines. To hold these 1,000 characters, the Commodore 128 uses a specific block of memory that is, not surprisingly, 1,000 bytes long. This block of memory, called screen memory, normally starts at memory address $400 and extends to memory address $7E7 in memory bank 0. You can visualize it as a grid of squares measuring 40 columns wide by 25 lines high, with each square representing one screen location. Figure 11-1 is a map of the C-128's 40-column screen.

Figure 11-1
C-128 screen memory map



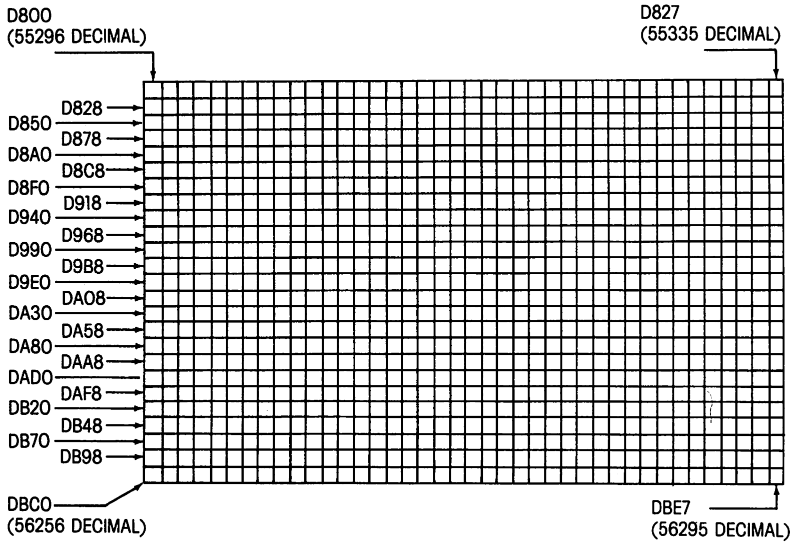
Here's how the BALLBOUNCE.BAS program works. When you type a character, the computer's operating system translates the character into a special code, and then prints the character on the screen by storing its code number in the appropriate screen memory location. The character codes used for this purpose are not the standard ASCII codes commonly used by computer printers and for computer-to-computer communications. Instead, the Commodore 128 uses a special set of screen codes that includes many special characters in addition to the standard set of ASCII characters. Listings of these screen codes can be found in appendix C.

When you know these screen display codes and the location of the screen display memory, you can print text and graphics characters on the computer's screen by poking the screen code values directly into the appropriate screen memory locations. In this way, you can bypass the computer's operating system and screen editor at any time, and can print anything you like directly on the screen!

C-128 Color Memory

In addition to its 1,000-byte block of screen memory, the Commodore 128 has a corresponding 1,000-byte block of color memory. This block of color RAM begins at memory location $D800 in bank 15 and extends to memory location $DBE7 in bank 15. Like the text screen map illustrated in figure 11-1, this bank of color memory can also be thought of as a 40-column-by-25-line matrix of squares, with each square representing one pixel (picture element) on your computer screen. The color memory map of the Commodore 128 is illustrated in figure 11-2.

Figure 11-2 C-128 color memory map



The C-128's screen memory map and color memory map are designed to be used together. In addition to its screen display codes, the Commodore 128 also has a set of 16 color codes. By poking those codes into the computer's color memory map, you can determine the color of each individual text or graphics character that appears on the screen. The color codes used by the Commodore 128 are shown in table 11-1.

Table 11-1
Commodore 128's Color Codes

Code	Color	Code	Color
0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Gray 1
4	Purple	12	Gray 2
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Gray 3

How Screen Maps and Color Maps Work Together

A close examination of figures 11-1 and 11-2 (the screen map and the color map of the Commodore 128) reveals how the C-128's screen memory and color memory work together. Because both maps have the same measurements—40 columns wide by 25 rows deep—the 16 colors that can be used on the color map can be visualized as a set of 16 colored overlays. By placing these overlays on the appropriate portions of your computer's screen map, you can make the characters on the grid appear in any color you choose.

Compare figures 11-1 and 11-2, and you'll see exactly how this color overlay concept is used in the BALLBOUNCE program. In line 40, a loop places a red overlay over the top two thirds of the screen—from the top line of the screen down to the line that begins at memory location 55976. When this red overlay is first laid down, it is invisible because nothing has been drawn on the screen yet. But as soon as something is printed on the portion of the screen covered by the overlay—for example, a bouncing ball—the character shows up in red, as you have just seen if you ran the program.

After the red overlay is in place, a yellow one is laid down. This yellow overlay is just one pixel high; it runs across the screen horizontally in the form of a horizontal line that extends from location 55976 to location 56015. Next, a line is drawn across the screen, and comes out yellow because it lies under our yellow overlay.

After the red and yellow overlays are in place, the words "FOLLOW THE BOUNCING BALL..." are printed across the top of the screen in white letters, using conventional PRINT commands. Then, in lines 210 through 240, a red ball is sent bouncing across the screen. This animation technique is quite simple; a circle (screen code 81) is drawn on the screen, then suddenly erased and redrawn in a new location. The square in which the ball appears keeps changing, and the effect is one of crude animation. When you consider that the program is written in BASIC, and that sprites (movable, arcade-style graphics characters) are not used, the animation in this little program is pretty effective. But a number of far better animation techniques are available to assembly language programmers, as you'll discover later in this volume.

Controlling Animation with Hand Controllers

The next program we'll look at is a BASIC program called JOYSTICK.BAS, which is shown in listing 11-2. It also uses the C-128's 40-column text mode, and contains some very important principles that are applicable to graphics programs written in assembly language. So don't skip over it, either! Type and run the JOYSTICK program; it will help you understand the assembly language programs that follow.

Listing 11-2
JOYSTICK.BAS
program

```
10 REM ***** JOYSTICK.BAS **********
20 PRINT CHR$(147):REM CLEAR SCREEN
30 BASE=1024:REM START OF SCREEN MEMORY
40 X=INT(40/2):REM POSITION X HALFWAY
   ACROSS SCREEN
50 Y=INT(25/2):REM POSITION Y HALFWAY DOWN
   SCREEN
60 POKE 53280,4:POKE 53281,0:REM PURPLE
   BORDER, BLACK BACKGROUND
70 FOR L=55296 TO 56295:REM COLOR MAP
80 POKE L,7:NEXT L:REM YELLOW CHARACTERS
90 REM **** READ JOYSTICK ****
100 POKE BASE+X+40*Y,81:REM PRINT DOT AT
    SCREEN POSITION X,Y
110 JV=PEEK(56320):REM GET JOYSTICK VALUE
120 TB=JV AND 16:REM GET TRIGGER BUTTON
    STATUS
130 JV=15-(JV AND 15):REM CONVERT SWITCH
    VALUES TO AN NR BETWEEN 0 AND 10
140 IF TB=16 THEN POKE BASE+X+40*Y,32:REM
    IF TB NOT PRESSED, PRINT SPACE
150 IF JV<>0 THEN 170:REM JOYSTICK HAS BEEN
    ACTIVATED; READ IT
160 GOTO 100
170 ON JV GOTO 1100,1200,1300,1400,1500,
    1600,1700,1800,1900,2000
1000 REM ******** READ POINTER *********
1100 Y=Y-1:IF Y<0 THEN Y=24:REM UP
1110 GOTO 100:REM PRINT PIXEL
1200 Y=Y+1:IF Y>24 THEN Y=0:REM DOWN
1210 GOTO 100
1300 GOTO 100:REM NO ACTION
1400 X=X-1:IF X<0 THEN X=39:REM LEFT
1410 GOTO 100
1500 X=X-1:IF X<0 THEN X=39:REM LEFT...
1510 Y=Y-1:IF Y<0 THEN Y=24:REM AND UP
```

Listing 11-2 cont.

```
1520 GOTO 100
1600 X=X-1:IF X<0 THEN X=39:REM LEFT...
1610 Y=Y+1:IF Y>24 THEN Y=0:REM AND DOWN
1620 GOTO 100
1700 GOTO 100:REM NO ACTION
1800 X=X+1:IF X>40 THEN X=0:REM RT...
1810 GOTO 100
1900 X=X+1:IF X>40 THEN X=0:REM RT...
1910 Y=Y-1:IF Y<0 THEN Y=24:REM AND UP
1920 GOTO 100
2000 X=X+1:IF X>40 THEN X=0:REM RT...
2010 Y=Y+1:IF Y>24 THEN Y=0:REM AND DOWN
2020 GOTO 160
```

To run the program, you'll need a hand controller: a joystick or, even better, a trackball or a mouse. Plug the controller into Joystick Port A, load the program into RAM, and type RUN. After a few moments, a flickering yellow dot appears in the middle of the screen.

Move your joystick, and the yellow dot moves around the screen. Press the joystick's trigger button while you're moving the joystick, and the yellow dot leaves a trail of dots as it moves. When you move the dot over other dots, without pressing the trigger button, the other dots are erased. Try it, and watch the show!

Here's how the JOYSTICK.BAS program works. In line 3, an important constant called BASE is defined. The value of this constant, 1024 ($400 in hexadecimal notation), is the default starting address of the Commodore 128's low-resolution screen map. In lines 40 and 50, two variables (called X and Y) are set up for use as screen coordinates. Their initial values are defined as INT(40/2) for X and INT (25/2) for Y. Because the Commodore's low-resolution screen measures 40 columns wide by 25 rows high, these values print a dot at midscreen when the program begins.

Line 60 sets the border and background screen colors for the JOYSTICK.BAS screen display. Then, in lines 70 and 80, the value 7 is stored in every byte of RAM in the C-128's color map, which extends from $D800 to $DEB7 in bank 15. This block fill operation ensures that all the characters we'll be printing on the screen are yellow.

As we saw in the BALLBOUNCE.BAS program, the C-128's color map can be thought of as a color overlay. When a color is stored in a pixel on this overlay, the corresponding pixel on the Commodore screen map is displayed in the chosen color. Because the number 7 is the Commodore 128's code for the color yellow (for a list of all color codes, see table 11-1), storing a 7 in every byte of memory from $D800 through $DBE7 makes every character on the screen yellow; hence, what you'll get in the JOYSTICK.BAS program is yellow dots.

Reading a Hand Controller

The next section of the JOYSTICK.BAS program reads the hand controller in Port A. The Commodore 128 has a pair of joystick ports that are often referred to as Port A and Port B. The status of Port A can be determined by peeking into an 8-bit memory register located at memory address $DC00 (or 56320 in decimal notation). The status of Port B can be determined by peeking into a similar memory register at memory address $DC01 (decimal 56321).

Each of the two joysticks that can be plugged into the Commodore 128 contains five on/off switches. Four of these switches correspond to the four primary directions in which a joystick can be pushed: up, down, left, and right. If the joystick is moved diagonally, two switches are activated simultaneously and read in combination. In this way, diagonal movements of the joystick are detected. The fifth switch inside a joystick determines whether the device's trigger button is pressed or not pressed.

The JOYSTICK.BAS program is designed to read a hand controller plugged into Port A of the Commodore 128 console. The program reads the joystick by peeking into memory address $DC00 (56320). Table 11-2 shows all possible values that can be found in that location, and their meanings.

Table 11-2 C-128 Hand Controller Values

Switch Value	Binary Value	Meaning
0	0000 0000	No action
1	0000 0001	Up
2	0000 0010	Down
3	0000 0011	None
4	0000 0100	Left
5	0000 0101	Left + up
6	0000 0110	Left + down
7	0000 0111	None
8	0000 1000	Right
9	0000 1001	Right + up
10	0000 1010	Right + down
11	0000 1011	None
12	0000 1100	None
13	0000 1101	None
14	0000 1110	None
15	0000 1111	None
16	0001 0000	Trigger button pressed
17	0001 0001	Trigger + up
18	0001 0010	Trigger + down
19	0001 0011	None
20	0001 0100	Trigger + left
21	0001 0101	Trigger + left + up
22	0001 0110	Trigger + left + down
23	0001 0111	None
24	0001 1000	Trigger + right
25	0001 1001	Trigger + right + up
26	0001 1010	Trigger + right + down
>26	>0001 1010	None

Checking the Trigger Button

After the flashing dot is placed in the center of the screen in the JOYSTICK.BAS program, a loop is set up to determine the value of the joystick's direction switches. But before the direction of the joystick is read, a test is conducted to determine whether the trigger button is being pressed. If the trigger button is being pressed, a small circle is drawn on the screen at the current joystick position. If the trigger button is not being pressed, no circle is drawn and a space is printed on the screen to erase any circle that may have been printed in that location.

The result of all this plotting, drawing, and erasing of tiny circles is quite entertaining. By pressing a joystick trigger and moving the joystick around, you can draw patterns of circles all over the computer screen. By moving the joystick and not pressing the trigger button, you can move the cursor without drawing any patterns, and you can also erase any patterns that may lie in your path.

When BASIC Is Enough

The JOYSTICK program works so well in BASIC that there's no compelling reason for translating it into assembly language. In fact, if it were converted to assembly language, it would run too fast! (I did translate it into assembly language, and the results were very disappointing. Each time I moved the joystick, long strings of dots instantly appeared on the screen, instead of moving as they did in the BASIC version of the program—across the screen in a controlled manner. You can perform the same frustrating experiment, if you're interested.)

BASIC is not such a great language, though, for writing high-resolution graphics programs. High-resolution programs have to be written using an extremely complicated technique called bit mapping, and BASIC is far too slow to handle bit-mapping routines efficiently. The next program, called MAKEWAVE.BAS, clearly demonstrates how unbearably slow a high-resolution program written in BASIC can be. But as you sit and wait for the program to crawl through its paces, don't get too impatient. Later, you'll get an opportunity to type, assemble, and execute an assembly language version of the same program—and I guarantee that you'll notice the change!

First, though, type the MAKEWAVE.BAS program, which is shown in listing 11-3. You'll then see very clearly why BASIC is not considered the best language for writing high-resolution graphics programs, and why fast action arcade-style computer games are almost always written in assembly language.

Listing 11-3
MAKEWAVE.BAS program

```
10 REM *** MAKEWAVE.BAS ***
20 COLOR 0,1:COLOR 4,1:GRAPHIC 1,1
30 BANK 0:POKE 2604,120:BANK 15:REM PUT BIT
   MAP AT $2000, COLOR MAP AT $1C00
```

Listing 11-3 cont.

```
40 POKE 216,32:REM SET BIT-MAP FLAG
50 POKE 53265,PEEK(53265)OR 32:REM ENABLE
   BIT-MAP MODE
60 BASE=8192:REM START BIT MAP AT $2000
70 REM *** DRAW BASE LINE ***
80 Y=100:REM PLACE Y AXIS AT MIDSCREEN
90 FOR X=0 TO 319:REM DRAW X AXIS
100 GOSUB 170:NEXT X
110 REM *** DRAW SINE WAVE ***
120 FOR X=0 TO 319 STEP .5
130 Y=INT(100+80*SIN(X/10))
140 GOSUB 170:NEXT X
150 GOTO 150
160 FOR X=0 TO 319: REM DRAW X AXIS
170 COL=INT(X/8)
180 ROW=INT(Y/8)
190 LINE=Y AND 7
200 BYTE=BASE+ROW*320+8*COL+LINE
210 BITT=7-(X AND 7)
220 POKE BYTE,PEEK(BYTE) OR (2↑BITT)
230 RETURN
```

Modes in the C-128

The Commodore 128, as pointed out in previous chapters, has three main display modes: 40-column text mode, 80-column text mode, and 40-column (320-by-200 dot) high-resolution graphics mode. Because the 128's high-resolution mode has the same amount of dot-for-dot resolution as its 40-column text mode, it is sometimes referred to as the computer's 40-column high-resolution mode. The C-128's 80-column mode was designed primarily for text, so it is not usually referred to as a high-resolution mode. The C-128 does have a multicolor graphics mode though, and there's a multicolor text mode, too, but we won't be discussing either of those modes in this chapter.

40-Column Modes

When the C-128 is in 40-column text mode, it displays 25 lines of 40 typed characters each (a total of 1,000 characters) on its monitor screen. Each of these characters is made up of eight bytes of binary data.

In its 40-column high-resolution mode, the Commodore 128 produces a screen display 320 dots (or pixels) wide and 200 pixels high. That's a total of 64,000 separate dots, each of which requires one bit of memory. So it takes 8,000 bytes of memory to produce a high-resolution screen display.

Text Modes

When the C-128 is in 40-column or 80-column text mode, the characters used to create its text display are stored in memory as binary data. It takes eight bits of data to create one character. When the eight bits of data that form a character's image are displayed on the screen, they are arranged as shown in table 11-3.

Table 11-3 Displaying a Character on the Screen

Binary Notation	Hexadecimal Notation	Appearance
00000000	00	
00011000	18	XX
00111100	3C	XXXX
01100110	66	XX XX
01100110	66	XX XX
01111110	7E	XXXXXX
01100110	66	XX XX
00000000	00	

Drawing Characters on the Screen

When the Commodore 128 is in text mode and a character is shown on the screen, a code number representing that character is stored in the C-128's screen map—a block of RAM designated as screen memory. Each time the C-128's VIC-II graphics chip creates a screen display (and it does it 60 times every second), it fetches each character code stored in the screen memory, and uses the code as a pointer to another block of memory called character generator ROM. In character generator ROM, a 64-bit image of every character in the Commodore 128's character set is stored as a series of eight bits. These eight bits are what the VIC-II chip uses to create the characters that it displays on the Commodore 128's video screen.

Setting up a Bit-Mapped Display

When the C-128 is in one of its high-resolution (bit-mapped) modes, it doesn't use the preprogrammed characters stored in character generator ROM. Instead, each individual dot on its video screen map is represented by one bit of data in the block of RAM that's used as a high-resolution screen map. So, if you know the exact position of a dot on the screen, you can turn that dot off or on by simply setting or clearing its corresponding bit in video memory. In this way, you can control every dot on the screen.

This would make bit mapping a very simple matter if a dot could be plotted on the Commodore's screen using simple X and Y coordinates. Unfortunately, that's not the way high-resolution bit mapping works on the Commodore 128. The dots that make up the Commodore's screen display don't go straight across and down the screen as they do in text mode. Instead, they're arranged as if they were dots in text characters, in 8-by-8 dot matrixes. These matrixes are placed on the screen in a 40-column-by-25-line configuration, as if they were standard text characters. This arrangement produces a high-resolution screen display that measures 320 dots (pixels) wide by 200 dots (pixels) high. That's a total of 64,000 dots, each one of which can be individually turned on and off.

To illustrate how this works, table 11-4 shows where a Commodore 128 gets the data that it uses for the first two rows of data on a high-resolution screen.

Table 11-4
How the C-128 Produces a Bit-Mapped Display

	Column 1	Column 2	Column 3	...	Column 40
Line 1	Byte 0	Byte 8	Byte 16	...	Byte 312
Line 2	Byte 1	Byte 9	Byte 17	...	Byte 313
Line 3	Byte 2	Byte 10	Byte 18	...	Byte 314
Line 4	Byte 3	Byte 11	Byte 19	...	Byte 315
Line 5	Byte 4	Byte 12	Byte 20	...	Byte 316
Line 6	Byte 5	Byte 13	Byte 21	...	Byte 317
Line 7	Byte 6	Byte 14	Byte 22	...	Byte 318
Line 8	Byte 7	Byte 15	Byte 23	...	Byte 319
Line 9	Byte 320	Byte 328	Byte 336	...	Byte 632
Line 10	Byte 321	Byte 329	Byte 337	...	Byte 633
Line 11	Byte 322	Byte 330	Byte 338	...	Byte 634
Line 12	Byte 323	Byte 331	Byte 339	...	Byte 635
Line 13	Byte 324	Byte 332	Byte 340	...	Byte 636
Line 14	Byte 325	Byte 333	Byte 341	...	Byte 637
Line 15	Byte 326	Byte 334	Byte 342	...	Byte 638
Line 16	Byte 327	Byte 335	Byte 343	...	Byte 639
...and so on.					

Doing It with Equations

In the zigzag layout illustrated in table 11-4, it's easy to mix text and bit-mapped graphics on the Commodore 128, because text and high-resolution graphics are laid out on the screen in the same way. Unfortunately, it also makes the job of bit mapping the C-128 screen rather complicated. To map a dot on a Commodore 128 high-resolution display, you have to use a fairly complex mathematical formula. First, you have to figure out where the dot lies on a 320-by-200 square grid, using a pair of variables (which I'll call X and Y) for the grid's column (X) and row (Y) coordinates. Then, because the C-128 screen map is subdivided into 8-by-8 dot matrixes, you have to break the screen map down into 8-by-8 dot subdivisions by dividing each coordinate by 8. In the MAKEWAVE.BAS program, this task is accomplished in lines 170 and 180, as follows:

```
170 COL=INT(X/8)
180 ROW=INT(Y/8)
```

Next, you have to figure out the dot's coordinates inside its 8-by-8 dot matrix. Here's how the MAKEWAVE.BAS program does it:

```
190 LINE=Y AND 7
```

```
200 BYTE=BASE+ROW*320+COL*8+LINE
```

Finally, you can turn on the bit you have selected with a line such as this:

```
220 POKE BYTE,PEEK(BYTE) OR (2↑BIT)
```

The multiple steps in this formula take a long time to process in BASIC—and that's one reason why high-resolution graphics programs written in BASIC run so slowly. Fortunately, as we shall soon see, the calculation takes much less time in assembly language.

High-Resolution Color Graphics

Now let's take some time out to see how the Commodore 128 generates the screen colors that it uses to produce a high-resolution screen display. It is important to point out that the colors used in bit-mapped C-128 graphics do not come from the 1,000-byte text and low-resolution color map that begins at memory address $D800 in block 15. Instead, they come from a special high-resolution color map that extends from $1C00 to $1CFF.

When the C-128 is in 320-by-200 dot bit-mapped mode, the upper four bits of each byte stored in this block of memory define the color of any bit that is cleared to 1 in a corresponding 8-by-8 dot area on the screen. The lower four bits of each block define the color of any bit that is cleared to 0 in that same 8-by-8 block.

C-128 Memory Map: A Review

Shortly, we'll be examining MAKEWAVE.BAS in more detail. But first, it might be beneficial to review the memory architecture of the C-128. As explained in chapter 10, 128K of RAM and almost 48K of ROM are installed in the C-128. To help the programmer address all of this memory, the computer is equipped with 15 preset memory configurations called banks. Of these 15 banks, there are 4 that are of paramount importance: banks 0, 1, 14, and 15.

Banks 0 and 1, as you may recall, are RAM banks. When the C-128 is running a BASIC program, the computer ordinarily stores the program's text in bank 0, and places a table of the variables which the program uses in bank 1. Banks 14 and 15 are primarily ROM banks. The C-128's BASIC interpreter resides in bank 15, and the data used to generate screen characters is stored in bank 14.

When a BASIC program is stored in bank 0, the normal location for BASIC programs, its text ordinarily starts at memory address $1C00. However, as you may have noticed if you examined the memory maps in appendix F, the block of memory that starts at $1C00 is also used as screen memory when the C-128 is in 40-column high-resolution mode.

Because a BASIC program and a high-resolution screen can't occupy the same RAM space at the same time, the engineers who designed the C-128 provided a handy technique for keeping BASIC programs and high-resolution screen data out of each other's way. But the technique works only if the C-128 is switched to high-resolution mode using a BASIC 7.0 GRAPHIC command. If a GRAPHIC command is issued to put the C-128 into high-resolution mode, and a BASIC program is in bank 0 RAM when the command is received, the GRAPHIC command automatically moves the BASIC program up from its normal starting address of $1C00 to a new starting address of $4000. And that's where the program will stay, even if another GRAPHIC command is issued to put the computer back into 40-column text mode.

MAKEWAVE.BAS Program, Line by Line

Now let's take a close look at the MAKEWAVE.BAS program, beginning with line 20. The first two commands in this line—COLOR 0,1 and COLOR 4,1—set the color of the screen display. Then, the command GRAPHIC 1,1 sets up (and clears the screen for) a high-resolution display. As explained previously, this command also moves the text of the MAKEWAVE.BAS program up to address $4000, so the program will still be in memory and will still be executable when the computer has entered its high-resolution mode.

In line 30 of the MAKEWAVE.BAS program, a BANK 0 command takes the C-128 out of bank 15—the "home" bank for executing BASIC programs—and puts the computer temporarily in bank 0, the RAM bank in which screen memory resides.

When the switch to bank 0 has been carried out, a POKE instruction places the value 120 (or $78 in hexadecimal notation) into memory location 2604 (or $A2C in hex). Memory address $A2C, as noted in chapter 10, is a "shadow register" used to access another important address: $D018, the C-128's VMCSB register. In the Commodore 128, and in the Commodore 64, the setting of the VMCSB register determines where the VIC-II chip looks to find the data needed to create text and high-resolution screens. When the C-128 is in high-resolution mode, the four low bits of the VMCSB register tell the VIC-II chip where screen memory begins, and the four high bits tell the VIC-II where it can find the data it needs to determine what colors should be displayed on the screen. So the POKE command in line 30 of the MAKEWAVE.BAS program notifies the C-128's VIC-II chip, through memory address $A2C, that it can find a screen map beginning at $2000 and a color map starting at $1C00 (the normal starting address for color maps in C-128 high-resolution graphics programs).

After the value $78 is poked into memory address $A2C, the C-128 is returned to bank 15 so that it can run the rest of the MAKEWAVE.BAS program. Then, in line 40, a very important operation occurs; the value 32 ($20 in hex) is poked into memory address

216 ($D8 in hex notation). This operation may not sound familiar to Commodore 64 programmers because it has no equivalent in C-64 programming. But it is of critical importance in high-resolution C-128 programs such as MAKEWAVE.BAS. Here's why.

In the C-128 (but *not* in the C-64), memory location 216 (or $D8 in hex) is a flag that determines what kind of display the computer will generate in 40-column mode. Every 1/60 of a second, the C-128 checks memory address $D8 and immediately goes into the graphics mode indicated by the flag's setting. And, because the register's default setting is for 40-column text, the C-128 will not stay in high-resolution graphics mode for more than 1/60 of a second unless the default value of memory location $D8 is changed. The settings of the flag are shown in table 11-5.

Table 11-5
Setting the $D8 Register

Decimal	Hexadecimal	Mode
224	$E0	GRAPHIC 4 (split screen, multicolor high resolution and text)
160	$A0	GRAPHIC 3 (multicolor, high resolution)
96	$60	GRAPHIC 2 (split screen, high resolution and text)
32	$20	GRAPHIC 1 (high resolution)
0	$00	GRAPHIC 0 (text)

SCROLY Register

The POKE instruction in line 50 is also quite important; but, unlike the POKE in the previous line, this one is also used in Commodore 64 high-resolution programs. It sets bit 4 of memory address 53265 ($D011), an important C-64/C-128 register called the SCROLY register. In the Commodore 128, and in the Commodore 64, bit 4 of the SCROLY register is what turns on the computer's bit-mapped 40-column mode.

Now we have come to line 60 of MAKEWAVE.BAS—and from that line on, every instruction in the MAKEWAVE.BAS program would be just as much at home in a Commodore 64 program as it is in this one. In line 60, a BASIC variable called BASE is defined, and its value is set at 8192 (or $2000 in hex). This is the starting point of the high-resolution screen map set up in line 30. In statements 70 through 100, a horizontal line is drawn across the middle of the screen using a standard, screen-plotting subroutine that extends from line 170 through line 230. Next, in lines 110 through 140, a sine wave is drawn on the screen using the screen-plotting subroutine in lines 170 through 230 and the standard BASIC function SIN(X). The program ends with an infinite loop in line 150.

In Search of a Faster Program

Because the MAKEWAVE.BAS program is written completely in BASIC, it runs quite slowly. One way to improve its speed might be to convert the screen-plotting subroutine into an assembly language program. Then the subroutine could be assembled into machine language and called from BASIC each time it is needed. This is the approach taken in listings 11-4 and 11-5: a BASIC program called MAKEWAVE2.BAS and an assembly language program called PLOTWAVE.S, written using a Merlin 128 assembler.

Listing 11-4
MAKEWAVE2.BAS program

```
10 REM *** MAKEWAVE2.BAS ***
20 COLOR 0,1:COLOR 4,1:GRAPHIC 1,1
30 HPSN=DEC("0B02"):VPSN=DEC("0B04")
40 IF A=0 THEN A=1:BLOAD "PLOTWAVE.O"
50 HI=INT(4864/256):LO=4864-HI*256:REM
   ADDRESS OF 'PLOT' ROUTINE
60 POKE 4633,LO:POKE 4634,HI:REM SET USR(X)
   POINTERS
70 REM *** DRAW BASE LINE ***
80 Y=100:HI=INT(Y/256):LO=Y-HI*256
90 POKE VPSN,LO:POKE VPSN+1,HI
100 FOR X=0 TO 319:HI=INT(X/256):LO=X-HI*
    256
110 POKE HPSN,LO:POKE HPSN+1,HI
120 B=USR(C):NEXT X
130 REM *** DRAW SINE WAVE ***
140 FOR X=0 TO 319 STEP .5
150 HI=INT(X/256):LO=X-HI*256
160 POKE HPSN,LO:POKE HPSN+1,HI
170 Y=INT(100+80*SIN(X/10))
180 HI=INT(Y/256):LO=Y-HI*256
190 POKE VPSN,LO:POKE VPSN+1,HI
200 B=USR(C):NEXT X
210 GOTO 210
```

Listing 11-5
PLOTWAVE.S program

```
*
* PLOTWAVE.S
*
          ORG    $1300
*
HMAX      EQU    320
BASE      EQU    $2000
*
TEMPA     EQU    $FA
TEMPB     EQU    TEMPA+2
*
TABSIZ    EQU    $0B00
*
HPSN      EQU    TABSIZ+2
VPSN      EQU    HPSN+2
CHAR      EQU    VPSN+1
```

Listing 11-5 cont.

```
ROW       EQU   CHAR+1
LINE      EQU   ROW+1
BYTE      EQU   LINE+1
BITT      EQU   BYTE+2
*
MPRL      EQU   BITT+1
MPRH      EQU   MPRL+1
MPDL      EQU   MPRH+1
MPDH      EQU   MPDL+1
PRODL     EQU   MPDH+1
PRODH     EQU   PRODL+1
*
          JMP   START
*
* BLOCK FILL ROUTINE
*
* 16-BIT MULTIPLICATION
*
MULT16    LDA   #0
          STA   PRODL
          STA   PRODH
          LDX   #17
          CLC
MULT      ROR   PRODH
          ROR   PRODL
          ROR   MPRH
          ROR   MPRL
          BCC   CTDOWN
          CLC
          LDA   MPDL
          ADC   PRODL
          STA   PRODL
          LDA   MPDH
          ADC   PRODH
          STA   PRODH
CTDOWN    DEX
          BNE   MULT
          RTS
*
* PLOT ROUTINE
*
* 8-BIT DIVISION
* (ROW=VPSN/8)
*
START     LDA   VPSN
          LSR   A
          LSR   A
          LSR   A
          STA   ROW
```

Listing 11-5 cont.

```
*
* CHAR=HPSN/8
*
         LDA   HPSN
         STA   TEMPA
         LDA   HPSN+1
         STA   TEMPA+1
         LDX   #3
DLOOP    LSR   TEMPA+1
         ROR   TEMPA
         DEX
         BNE   DLOOP
         LDA   TEMPA
         STA   CHAR
*
* LINE=VPSN AND 7
*
         LDA   VPSN
         AND   #7
         STA   LINE
*
* BITT=7-(HPSN AND 7)
*
         LDA   HPSN
         AND   #7
         STA   BITT
         SEC
         LDA   #7
         SBC   BITT
         STA   BITT
*
* FORMULA TO PLOT DOT
*
* MULTIPLY ROW * HMAX
*
         LDA   ROW
         STA   MPRL
         LDA   #0
         STA   MPRH
         LDA   #<HMAX
         STA   MPDL
         LDA   #>HMAX
         STA   MPDH
         JSR   MULT16
         LDA   MPRL
         STA   TEMPA
         LDA   MPRL+1
         STA   TEMPA+1
*
```

Listing 11-5 cont.

```
* ADD PRODUCT TO BASE
*
          CLC
          LDA   #<BASE
          ADC   TEMPA
          STA   TEMPA
          LDA   #>BASE
          ADC   TEMPA+1
          STA   TEMPA+1
*
* MULTIPLY 8 * CHAR
*
          LDA   #8
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   CHAR
          STA   MPDL
          LDA   #0
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
          STA   TEMPB
          LDA   MPRH
          STA   TEMPB+1
*
* ADD LINE
*
          CLC
          LDA   TEMPB
          ADC   LINE
          STA   TEMPB
          LDA   TEMPB+1
          ADC   #0
          STA   TEMPB+1
*
* TEMPA + TEMPB = BYTE
*
          CLC
          LDA   TEMPA
          ADC   TEMPB
          STA   TEMPB
          LDA   TEMPA+1
          ADC   TEMPB+1
          STA   TEMPB+1
*
* BYTE=BYTE OR 2↑BIT
*
          LDX   BITT
```

Listing 11-5 cont.

```
          INX
          LDA   #0
          SEC
SQUARE    ROL
          DEX
          BNE   SQUARE
          LDY   #0
          ORA   (TEMPB),Y
          STA   (TEMPB),Y
          RTS
*
```

PLOTWAVE.S and MAKEWAVE2.BAS are designed to be used together. Each time PLOTWAVE.S is called by MAKEWAVE2.BAS, it plots a dot on the screen. Before PLOTWAVE.S is called, however, the horizontal screen coordinate of the dot to be plotted must be stored in memory addresses $0B02 and $0B03, and the vertical coordinate of the dot must be placed in memory address $0B04. Then PLOTWAVE.S can be called from BASIC using BASIC's USR(X) function.

PLOTWAVE.S is an assembly language version of the BASIC dot-plotting subroutine that appears in lines 160 through 230 of the MAKEWAVE2.BAS program.

USR(X) Function

One noteworthy feature of the MAKEWAVE2.BAS program is the way it uses the USR(X) function, which was introduced in chapter 5. As you may remember from chapter 5, there are some differences between the way the USR(X) function is used in Commodore 128 programs and the way it is used in Commodore 64 programs. Before USR(X) is used in a C-64 BASIC program, the starting address of the machine language program that it calls must be placed in memory registers 785 and 786 ($0311 and $0312 in hex notation). In programs written for the C-128, however, the address of the machine language program must be placed in memory locations 4633 and 4634 ($1219 and $1220 in hex notation).

In line 30 of the MAKEWAVE2.BAS program, with the help of the BASIC function DEC("X"), a pair of BASIC variables called HPSN (for "horizontal position") and VPSN (for "vertical position") are defined. These variables are set to point to memory addresses $0B02 and $0B04, the addresses where the PLOTWAVE.S program expects to find its horizontal and vertical screen coordinates when it is told to plot a dot on the screen.

In line 40 of the MAKEWAVE2.BAS program, a binary program called PLOTWAVE.O (the object code version of the PLOTWAVE.S program) is loaded into memory using a standard C-128 technique. First, a variable called A, which initially holds a value of 0, is changed to contain the value 1. Next, an IF...THEN statement loads PLOTWAVE.O into memory. PLOTWAVE.O will not load, however, unless the value of A is 0. This technique keeps PLOTWAVE.O from being loaded into memory over and over again, hanging up the C-128 at line 40 of the MAKEWAVE2.BAS program.

The construction used in lines 50 and 60—and also in several other lines in the MAKEWAVE2.BAS program—is another common feature of Commodore BASIC programs that interact with machine language programs. In these two lines, a standard BASIC algorithm loads USR(X) pointers 4633 and 4634 with the low and high bytes, respectively, of the address of the PLOTWAVE.O program. At several other places in the program, the same algorithm places other high byte/low byte combinations into other memory locations.

Although MAKEWAVE.BAS and MAKEWAVE2.BAS look quite different, they operate in a similar fashion. The main difference between them is that MAKEWAVE.BAS plots its dots in BASIC, and MAKEWAVE2.BAS plots its dots by calling the machine language program PLOTWAVE.O.

After you have typed and assembled PLOTWAVE.S, and typed and saved MAKEWAVE2.BAS, you can execute both programs with a single RUN command. Although MAKEWAVE2.BAS runs faster than its predecessor MAKEWAVE.BAS, it won't run as fast as you may have hoped. That's because the program contains a lot of time-consuming floating-point operations that are performed in BASIC—a process that, as you may know, is notoriously slow. So there's still too much BASIC in MAKEWAVE2.BAS to allow the program to operate much faster than its predecessor.

There is, however, one way to make a dot-plotting program run much faster than either MAKEWAVE.BAS or MAKEWAVE2.BAS. That method is to forget about BASIC altogether and write the whole program in assembly language. And that's what we'll do in chapter 12.

12

Advanced Commodore 128 High-Resolution Graphics

Getting up to speed

With the right software, the Commodore 128 can understand many languages—BASIC, C, Logo, Forth, and dozens more. But over the years, only one language has been used to create a significant number of commercial-quality, high-resolution graphics programs. That language is—wouldn't you know it?—assembly language.

The reason is speed. In previous chapters, we saw how painfully slow BASIC can be when it handles a graphics program—particularly a high-resolution graphics program. In this chapter, you'll get a chance to type and run two graphics routines that are written completely in assembly language and—not surprisingly—run considerably faster than the BASIC programs presented in chapters 10 and 11.

High-Resolution Screen Map

A source code listing titled SQUARE.S is presented later in this chapter. It is actually two high-resolution programs in one; one routine fills a high-resolution screen with the color of your choice, and another draws a square on a high-resolution screen.

SQUARE.S, like the programs presented in chapter 11, is a bit-mapped program designed to run in the C-128's 40-column high-resolution mode. That means, as you may recall from chapter 11, that the program generates its screen display using a block of RAM called a high-resolution screen map. The bit map used in the SQUARE.S program extends from memory address $2000 to memory address $3FFF (or from 8192 to 16383 in decimal notation). When the C-128 is in 40-column high-resolution mode, each bit of data stored in this block of RAM controls one dot (or pixel) on the computer's screen. If a data bit stored in the C-128's screen map is turned off, then its corresponding screen dot is also turned off. If a screen map bit is turned on, its corresponding screen dot is also turned on.

Because the C-128's high-resolution screen is 320 dots wide by 200 dots high, a program—or a programmer—can exercise control over 64,000 separate dots by using bit-setting, bit-clearing, and bit-shifting techniques. That's a lot of control for a programmer to have over a screen display—and that's how high-resolution graphics got its name!

Because the C-128's 40-column screen is 320 dots wide by 200 dots deep, you can pinpoint the location of any dot on the screen using two coordinates: an X coordinate that represents the dot's horizontal position, and a Y coordinate that represents its vertical position. Unfortunately, though, there is only an indirect relationship between a dot's screen coordinates and its corresponding bit in screen memory. The 64,000 bits that make up the C-128's screen are laid out in a slightly different way than their corresponding bits in screen RAM. This quirk makes it somewhat difficult to bit map the C-128 screen.

Here's a brief explanation of the problem. The data stored in the C-128's screen memory is arranged in a very straightforward way—

one bit right after the other, in 8,000 consecutive bytes of screen RAM. But the C-128's high-resolution screen is arranged in quite a different manner; instead of being laid out in consecutive bytes, like screen RAM, it is split into a grid of 1,000 rectangles, each one 8 bytes high. This grid measures 40 rectangles wide by 20 rectangles deep—1,000 cells in all, arranged exactly like the characters on the C-128's 40-column text screen.

Table 12-1 illustrates the relationship between the screen memory of the Commodore 128 and the display that the data produces on the screen. It shows where the first 32 bytes of screen RAM starting at memory address $2000 are located when they are displayed on a high-resolution screen.

Table 12-1 How Data Is Displayed

	Column 1	Column 2	Column 3	Column 4
Line 0	$2000	$2008	$2010	$2018
Line 1	$2001	$2009	$2011	$2019
Line 2	$2002	$200A	$2012	$201A
Line 3	$2003	$200B	$2013	$201B
Line 4	$2004	$200C	$2014	$201C
Line 5	$2005	$200D	$2015	$201D
Line 6	$2006	$200E	$2016	$201E
Line 7	$2007	$200F	$2017	$201F

The text-oriented layout shown in table 12-1 makes it quite easy to display text on the C-128's screen, because each 8-by-8 dot character that appears on the screen can be fashioned from 64 consecutive bits of screen RAM. But it complicates the job of the programmer working in high-resolution graphics, because it eliminates the possibility of using straight X/Y coordinates to plot dots. Instead, the relationship between each dot on the screen and its corresponding bit in screen RAM must be painstakingly calculated, using what must be one of the most complicated algorithms in the world of high-resolution graphics programming.

Plotting a Dot on the Screen

To illustrate how this complex algorithm works, let's go ahead and devise a system of coordinates for a 320-by-200 dot high-resolution screen, using X to represent each of the 320 dots across the screen and Y to represent each of the 200 dots (or bytes) down the screen. This arrangement is illustrated in figure 12-1.

As figure 12-1 shows, there are 320 possible X coordinates on a high-resolution screen, ranging from 0 to 319. And there are 200 possible Y coordinates, ranging from 0 to 199. So an X coordinate and a Y coordinate, used together, can plot any dot on the screen. But, because the screen is actually divided into 1,000 matrices of 64 dots each, we must devise some kind of conversion formula to use these X and Y coordinates to access data in screen RAM.

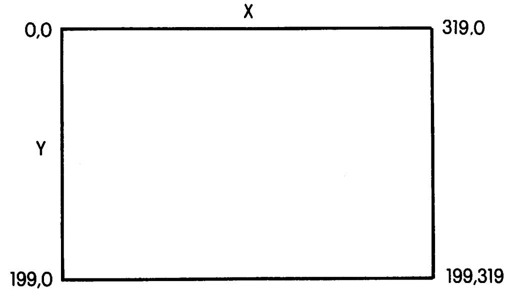


Figure 12-1 Using X/Y coordinates

The algorithm that is most often used for converting C-128 coordinates into screen memory addresses has several parts. First, because each rectangle on the screen is 8 dots wide by 8 dots high, both the X coordinate and the Y coordinate must be divided by 8. So, if we use the variable ROW to represent the starting address of a horizontal 8-byte row of dots, and the variable COL to represent the starting address of an 8-byte column of dots, we could start our algorithm with these two equations:

ROW = INT(Y/8)

COL = INT(X/8)

Next, because each horizontal row of dots is made up of 8 horizontal lines, we could number those lines 0 through 7 and find the line number of the dot in question with this equation:

LINE = Y AND 7

Another odd quirk about the C-128 screen is that the 8 bits in each byte of screen RAM are displayed in the opposite order from how they are stored in memory—with bit 0 on the left and bit 7 on the right. So we need an equation like the following to get the 8 bits in each byte of screen RAM into the proper order for a screen display:

BIT = 7 − (X AND 7)

After we determine the location of a bit in screen memory, we can add the base address of screen RAM to the equation. The sum should be the address of the byte in which the bit is situated. So let's join the preceding formulas and add them to the base address of the C-128's screen memory:

BYTE = ROW × 320 + COL × 8 + LINE + BASE

Finally, after the RAM address of a byte is calculated, the state of any given bit in that byte can be changed with a statement such as this:

POKE BYTE,PEEK(BYTE) OR 2↑BIT

To plot a dot on a high-resolution screen, you also need to understand the use of the C-128's color map, which begins at memory address $1C00 (7168 in decimal notation). The C-128 color map contains 1,000 bytes, and each byte determines the color of one 8-byte matrix on the screen. The upper 4 bits of each location in color memory define the color of each bit set to 1 in a corresponding 8-by-8 dot matrix on the screen. The lower 4 bits in each color map location define the color of any bit cleared to 0 in the same 8-by-8 matrix of pixels.

Because of the limitations of this system, the C-128 does not offer the programmer as much control over setting screen colors as it does over whether individual dots on the screen are off or on. Only two colors are available in each 8-by-8 dot (character size) matrix, and each dot in that matrix must be displayed in one of these two colors. However, there is no overall restriction on how many of the C-128's 16 colors can be displayed on the screen.

SQUARE.S Program

Now that we know how the C-128's screen map and color map work, we're ready to take a look at SQUARE.S, shown in listing 12-1. As mentioned previously, the program contains two separate routines. One, labeled PLOT, begins at line 97; the other, which draws a square on the screen, begins at line 258.

Listing 12-1
SQUARE.S program

```
 1 *
 2 * SQUARE.S
 3 *
 4           ORG    $1300
 5 *
 6 COLOR     EQU    $10
 7 BASE      EQU    $2000
 8 SCROLY    EQU    $D011
 9 BMPTR     EQU    $A2D
10 BMFLG     EQU    $D8
11 COLMAP    EQU    $1C00
12 *
13 HMAX      EQU    320
14 HSTART    EQU    105
15 VSTART    EQU    66
16 HEND      EQU    211
17 VEND      EQU    132
18 *
19 *
20 SCRLEN    EQU    8000
21 MAPLEN    EQU    1000
22 *
23 TEMPA     EQU    $FA
```

Listing 12-1 cont.

```
TEMPB     EQU   TEMPA+2
*
TABPTR    EQU   TEMPA
TABSIZ    EQU   $9000
*
HPSN      EQU   TABSIZ+2
VPSN      EQU   HPSN+2
CHAR      EQU   VPSN+1
ROW       EQU   CHAR+1
LINE      EQU   ROW+1
BYTE      EQU   LINE+1
BITT      EQU   BYTE+2
*
MPRL      EQU   BITT+1
MPRH      EQU   MPRL+1
MPDL      EQU   MPRH+1
MPDH      EQU   MPDL+1
PRODL     EQU   MPDH+1
PRODH     EQU   PRODL+1
*
FILVAL    EQU   PRODH+1
HPOS      EQU   FILVAL+1
*
          JMP   START
*
* BLOCK FILL ROUTINE
*
BLKFIL    LDA   FILVAL
          LDX   TABSIZ+1
          BEQ   PARTPG
          LDY   #0
FULLPG    STA   (TABPTR),Y
          INY
          BNE   FULLPG
          INC   TABPTR+1
          DEX
          BNE   FULLPG
PARTPG    LDX   TABSIZ
          BEQ   FINI
          LDY   #0
PARTLP    STA   (TABPTR),Y
          INY
          DEX
          BNE   PARTLP
FINI      RTS
*
* 16-BIT MULTIPLICATION
*
MULT16    LDA   #0
```

Listing 12-1 cont.

```
            STA   PRODL
            STA   PRODH
            LDX   #17
            CLC
MULT        ROR   PRODH
            ROR   PRODL
            ROR   MPRH
            ROR   MPRL
            BCC   CTDOWN
            CLC
            LDA   MPDL
            ADC   PRODL
            STA   PRODL
            LDA   MPDH
            ADC   PRODH
            STA   PRODH
CTDOWN      DEX
            BNE   MULT
            RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8
*
PLOT        LDA   VPSN
            LSR   A
            LSR   A
            LSR   A
            STA   ROW
*
* CHAR=HPSN/8
*
            LDA   HPSN
            STA   TEMPA
            LDA   HPSN+1
            STA   TEMPA+1
            LDX   #3
DLOOP       LSR   TEMPA+1
            ROR   TEMPA
            DEX
            BNE   DLOOP
            LDA   TEMPA
            STA   CHAR
*
* LINE+VSPN AND 7
*
            LDA   VPSN
            AND   #7
            STA   LINE
```

Listing 12-1 cont.

```
*
* BITT=7-(HPSN AND 7)
*
          LDA   HPSN
          AND   #7
          STA   BITT
          SEC
          LDA   #7
          SBC   BITT
          STA   BITT
*
* MULTIPLY ROW * HMAX
*
          LDA   ROW
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   #<HMAX
          STA   MPDL
          LDA   #>HMAX
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
          STA   TEMPA
          LDA   MPRL+1
          STA   TEMPA+1
*
* ADD PRODUCT TO BASE
*
          CLC
          LDA   #<BASE
          ADC   TEMPA
          STA   TEMPA
          LDA   #>BASE
          ADC   TEMPA+1
          STA   TEMPA+1
*
* MULTIPLY 8 * CHAR
*
          LDA   #8
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   CHAR
          STA   MPDL
          LDA   #0
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
```

Listing 12-1 cont.

```
                STA   TEMPB
                LDA   MPRH
                STA   TEMPB+1
*
* ADD LINE
*
                CLC
                LDA   TEMPB
                ADC   LINE
                STA   TEMPB
                LDA   TEMPB+1
                ADC   #0
                STA   TEMPB+1
*
* TEMPA + TEMPB = BYTE
*
                CLC
                LDA   TEMPA
                ADC   TEMPB
                STA   TEMPB
                LDA   TEMPA+1
                ADC   TEMPB+1
                STA   TEMPB+1
*
* BYTE=BYTE OR 2↑BIT
*
                LDX   BITT
                INX
                LDA   #0
                SEC
SQUARE          ROL
                DEX
                BNE   SQUARE
                LDY   #0
                ORA   (TEMPB),Y
                STA   (TEMPB),Y
                RTS
*
* MAIN ROUTINE
*
* DEFINE BIT MAP AND
* ENABLE HI-RES GRAPHICS
*
START           STA   $FF01
                LDA   #$78
                STA   BMPTR
*
                LDA   #$20
                STA   BMFLG
```

Listing 12-1 cont.

```
*
         LDA   #0
         STA   $FF00
         LDA   SCROLY
         ORA   #$20
         STA   SCROLY
         STA   $FF01
*
* CLEAR BIT MAP
*
         LDA   #0
         STA   FILVAL
         LDA   #<BASE
         STA   TABPTR
         LDA   #>BASE
         STA   TABPTR+1
         LDA   #<SCRLEN
         STA   TABSIZ
         LDA   #>SCRLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* SET BKG AND LINE COLORS
*
         LDA   #COLOR
         STA   FILVAL
         LDA   #<COLMAP
         STA   TABPTR
         LDA   #>COLMAP
         STA   TABPTR+1
         LDA   #<MAPLEN
         STA   TABSIZ
         LDA   #>MAPLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* DRAW HORIZONTAL LINES
*
         LDA   #VSTART
         STA   VPSN
         LDA   #<HSTART
         STA   HPSN
         LDA   #>HSTART
         STA   HPSN+1
         JSR   HDRAW
*
         LDA   #VEND
         STA   VPSN
         LDA   #<HSTART
```

Listing 12-1 cont.

```
                STA    HPSN
                LDA    #>HSTART
                STA    HPSN+1
                JSR    HDRAW
*
* DRAW VERTICAL LINES
*
                LDA    #VSTART
                STA    VPSN
                LDA    #<HSTART
                STA    HPOS
                LDA    #>HSTART
                STA    HPOS+1
                JSR    VDRAW
*
                LDA    #VSTART
                STA    VPSN
                LDA    #<HEND
                STA    HPOS
                LDA    #>HEND
                STA    HPOS+1
                JSR    VDRAW
*
INF             JMP    INF
*
HDRAW           JSR    PLOT
                INC    HPSN
                BNE    NEXT
                INC    HPSN+1
NEXT            LDA    HPSN+1
                CMP    #>HEND
                BCC    HDRAW
                LDA    HPSN
                CMP    #<HEND
                BCC    HDRAW
                RTS
*
VDRAW           LDA    HPOS
                STA    HPSN
                LDA    HPOS+1
                STA    HPSN+1
                JSR    PLOT
                INC    HPSN
                BNE    SKIP
                INC    HPSN+1
SKIP            JSR    PLOT
                LDX    VPSN
                INX
                STX    VPSN
```

Listing 12-1 cont.

```
318          CPX   #VEND
319          BCC   VDRAW
320          RTS
321  *
```

The SQUARE.S program was written using a Merlin 64 assembler. But with minor modifications, like all of the assembly language programs in this book, it can be typed and assembled using any other C-64 or C-128 assembler.

Type and assemble the SQUARE.S program, and store it on a disk as SQUARE.O. Then, the program can be loaded and executed by typing the BASIC 7.0 command:

```
BLOAD "SQUARE.O":SYS 4864
```

As you can see by looking at line 4, the SQUARE.S program assembles beginning at memory address $1300, or 4864 in decimal notation. So, if the program is executed using the command SYS 4864, it will start at line 47 and then jump to line 214, where the main part of the program begins. The program ends with an infinite loop (a loop that continues forever), so there is no elegant way to exit. This bug should be removed if SQUARE.S is expanded from a demonstration routine into a finished program.

Let's look at the main section of the program, which begins at line 214. This segment, using techniques that have been discussed in previous chapters, activates the C-128's high-resolution mode and clears the computer's screen map. Then, using a variable called COLOR, it fills the C-128's color map with the value $10, which produces a white foreground and a black background on the screen. (You can use a different color set by changing the value of the COLOR variable.)

The heart of the program is an assembly language dot-plotting routine labeled PLOT, which extends from line 97 to line 207. This module, as you can see by looking at the remarks in the source code, works like the dot-plotting routines that were included in the BASIC programs in chapters 10 and 11.

The program SQUARE.S, with the help of a Y register loop, uses the subroutine PLOT to fill the screen with color. Then the program, using vertical and horizontal lines with predetermined starting and ending points, calls the PLOT subroutine to draw a square on the screen.

When you type, assemble, and run the SQUARE.S program, you'll see that it runs fairly quickly, but not quite as quickly as it should—especially for an assembly language program. One reason why SQUARE.S runs more slowly than it should is that the PLOT subroutine calculates the complete address of each byte on the screen every time the byte is accessed. This procedure, as professional programmers discovered long ago, could be speeded up greatly with the help of a programming tool called a *Y lookup table.* A Y lookup table, as its name indicates, is a table that contains the starting address of each line, or Y coordinate, on a screen map. A Y lookup table contains only 200 addresses, because there are only 200 lines on the screen, so it can be created very quickly and then stored in memory. After a Y lookup table is created, it eliminates the need to look up Y addresses. Instead, when a program needs a Y coordinate address, it only has to look up the address using its Y lookup table.

The creation and use of Y lookup tables and a few other secrets of superfast graphics programming are explored—and demonstrated—in the next chapter.

13

Trade Secrets

How to write
a superfast graphics program

You know by now how fast assembly language is—and now we'll learn how to make it even faster. We'll reveal some of the secrets that professional programmers use when they want to write superfast assembly language programs.

In the past few chapters, we've seen how inadequate BASIC is as a tool for writing high-resolution graphics programs. In chapter 12, we translated one high-resolution BASIC program into assembly language and saw how much faster it ran. But that was just the beginning. Now we're going to improve the SQUARE.S program presented in chapter 12, and make it run even faster. By the time we're finished, we'll have it running at the speed of a commercial-quality assembly language program.

RECTANGLE.S Program

The program we'll be working with in this chapter, titled RECTANGLE.S, is shown in listing 13-1. It's an expanded version of the SQUARE.S program.

Listing 13-1
RECTANGLE.S
program

```
*
* RECTANGLE.S
*
         ORG    $1300
*
TEMPA    EQU    $FA
TEMPB    EQU    TEMPA+2
*
TABPTR   EQU    TEMPA
*
COLOR    EQU    $10
BMFLG    EQU    $D8
BMPTR    EQU    $A2D
COLMAP   EQU    $1C00
SCROLY   EQU    $D011
*
HMAX     EQU    320
*
SCRBAS   EQU    $2000
PTRL     EQU    $8000
PTRH     EQU    $8100
*
MAPLEN   EQU    1000
SCRLEN   EQU    8000
*
HSTART   EQU    $0C00
HEND     EQU    $0C02
VSTART   EQU    $0C04
VEND     EQU    $0C05
```

Listing 13-1 cont.

```
*
TABSIZ   EQU   VEND+1
HPSN     EQU   TABSIZ+2
VPSN     EQU   HPSN+2
CHAR     EQU   VPSN+1
ROW      EQU   CHAR+1
LINE     EQU   ROW+1
BYTE     EQU   LINE+1
BITT     EQU   BYTE+2
*
MPRL     EQU   BITT+1
MPRH     EQU   MPRL+1
MPDL     EQU   MPRH+1
MPDH     EQU   MPDL+1
PRODL    EQU   MPDH+1
PRODH    EQU   PRODL+1
*
FILVAL   EQU   PRODH+1
HPOS     EQU   FILVAL+1
*
         JMP   START
*
BITPSN   HEX   80,40,20,10,08,04,02,01
*
* BLOCK FILL ROUTINE
*
BLKFIL   LDA   FILVAL
         LDX   TABSIZ+1
         BEQ   PARTPG
         LDY   #0
FULLPG   STA   (TABPTR),Y
         INY
         BNE   FULLPG
         INC   TABPTR+1
         DEX
         BNE   FULLPG
PARTPG   LDX   TABSIZ
         BEQ   FINI
         LDY   #0
PARTLP   STA   (TABPTR),Y
         INY
         DEX
         BNE   PARTLP
FINI     RTS
*
* 16-BIT MULTIPLICATION
*
MULT16   LDA   #0
         STA   PRODL
```

Listing 13-1 cont.

```
                STA    PRODH
                LDX    #16
SHIFT           ASL    PRODL
                ROL    PRODH
                ASL    MPRL
                ROL    MPRH
                BCC    NOADD
                CLC
                LDA    MPDL
                ADC    PRODL
                STA    PRODL
                LDA    MPDH
                ADC    PRODH
                STA    PRODH
NOADD           DEX
                BNE    SHIFT
                RTS
*
* CREATE Y LOOKUP TABLE
*
MAKTAB          LDY    #0
YLOOP           CPY    #200
                BCC    CONT
                JMP    EXIT
*
* DIVIDE Y BY 8
*
CONT            TYA
                LSR    A
                LSR    A
                LSR    A
                STA    ROW
*
* MULTIPLY ROW * HMAX
*
                LDA    ROW
                STA    MPRL
                LDA    #0
                STA    MPRH
                LDA    #<HMAX
                STA    MPDL
                LDA    #>HMAX
                STA    MPDH
                JSR    MULT16
                LDA    PRODL
                STA    TEMPA
                LDA    PRODH
                STA    TEMPA+1
*
```

Listing 13-1 cont.

```
* ADD PRODUCT TO SCRBAS
*
          CLC
          LDA    #<SCRBAS
          ADC    TEMPA
          STA    PTRL,Y
          LDA    #>SCRBAS
          ADC    TEMPA+1
          STA    PTRH,Y
*
          INY
          JMP    YLOOP
*
EXIT      RTS
*
* MAIN ROUTINE
*
* DEFINE BIT MAP AND
* ENABLE HI-RES GRAPHICS
*
START     JSR    MAKTAB
*
          STA    $FF01
          LDA    #$78
          STA    BMPTR
*
          LDA    #$20
          STA    BMFLG
*
          LDA    #0
          STA    $FF00
          LDA    SCROLY
          ORA    #$20
          STA    SCROLY
          STA    $FF01
*
* CLEAR BIT MAP
*
          LDA    #0
          STA    FILVAL
          LDA    #<SCRBAS
          STA    TABPTR
          LDA    #>SCRBAS
          STA    TABPTR+1
          LDA    #<SCRLEN
          STA    TABSIZ
          LDA    #>SCRLEN
          STA    TABSIZ+1
          JSR    BLKFIL
```

Listing 13-1 cont.

```
177 *
178 * SET BKG AND LINE COLORS
179 *
180         LDA   #COLOR
181         STA   FILVAL
182         LDA   #<COLMAP
183         STA   TABPTR
184         LDA   #>COLMAP
185         STA   TABPTR+1
186         LDA   #<MAPLEN
187         STA   TABSIZ
188         LDA   #>MAPLEN
189         STA   TABSIZ+1
190         JSR   BLKFIL
191 *
192 * DRAW HORIZONTAL LINES
193 *
194         LDA   VSTART
195         STA   VPSN
196         LDA   HSTART
197         STA   HPSN
198         LDA   HSTART+1
199         STA   HPSN+1
200         JSR   HDRAW
201 *
202         LDA   VEND
203         STA   VPSN
204         LDA   HSTART
205         STA   HPSN
206         LDA   HSTART+1
207         STA   HPSN+1
208         JSR   HDRAW
209 *
210 * DRAW VERTICAL LINES
211 *
212         LDA   VSTART
213         STA   VPSN
214         LDA   HSTART
215         STA   HPOS
216         LDA   HSTART+1
217         STA   HPOS+1
218         JSR   VDRAW
219 *
220         LDA   VSTART
221         STA   VPSN
222         LDA   HEND
223         STA   HPOS
224         LDA   HEND+1
225         STA   HPOS+1
```

Listing 13-1 cont.

```
         JSR   VDRAW
*
INF      JMP   INF
*
HDRAW    JSR   PLOT
         INC   HPSN
         BNE   NEXT
         INC   HPSN+1
NEXT     LDA   HPSN+1
         CMP   HEND+1
         BCC   HDRAW
         LDA   HPSN
         CMP   HEND
         BCC   HDRAW
         RTS
*
VDRAW    LDA   HPOS
         STA   HPSN
         LDA   HPOS+1
         STA   HPSN+1
         JSR   PLOT
         INC   HPSN
         BNE   SKIP
         INC   HPSN+1
SKIP     JSR   PLOT
         LDX   VPSN
         INX
         STX   VPSN
         CPX   VEND
         BCC   VDRAW
         RTS
*
* CHAR=HPSN/8
*
PLOT     LDA   HPSN
         LSR   HPSN+1
         ROR
         LSR   HPSN+1
         ROR
         LSR   HPSN+1
         ROR
         STA   CHAR
*
* MULTIPLY 8 * CHAR
*
         LDA   #0
         ASL   CHAR
         ROL
         ASL   CHAR
```

Listing 13-1 cont.

```
         ROL
         ASL   CHAR
         ROL
         STA   TEMPB+1
*
* ADD LINE
*
         CLC
         LDA   VPSN
         AND   #7         ;LINE
         ADC   CHAR
         STA   TEMPB
         LDA   TEMPB+1
         ADC   #0
         STA   TEMPB+1
*
* BYTE = TEMPA + TEMPB
*
         CLC
         LDY   VPSN
         LDA   PTRL,Y
         ADC   TEMPB
         STA   TEMPB
         LDA   PTRH,Y
         ADC   TEMPB+1
         STA   TEMPB+1
*
* BYTE=BYTE OR 2↑BIT
*
         LDA   HPSN
         AND   #$07
         TAX
         LDY   #0
         LDA   (TEMPB),Y
         ORA   BITPSN,X
         STA   (TEMPB),Y
         RTS
*
```

The SQUARE.S program didn't do much; it merely drew a big empty square on a high-resolution screen. RECTANGLE.S, as its name implies, performs a similar function; it draws either a square or a rectangle on the screen, but it does the job much, much faster than its predecessor. In addition, you can control the shape, size, and location of the rectangle.

RECTANGLE.S, like SQUARE.S, was written using a Merlin 128 assembler. When the program has been typed, assembled, and saved on a disk, it can be called and executed using RECTANGLE.BAS, the BASIC program in listing 13-2.

Listing 13-2 RECTANGLE.BAS program

```
10 REM *** RECTANGLE.BAS ***
20 :
30 HST = 105
40 HND = 211
50 VST = 66
60 VND = 132
70 :
80 IF A=0 THEN A=1:BLOAD "RECTANGLE.O"
90 HI=INT (HST/256):LO=HST-HI*256
100 POKE DEC("0C00"),LO:POKE DEC("0C01"),HI
110 HI=INT (HND/256):LO=HND-HI*256
120 POKE DEC("0C02"),LO:POKE DEC("0C03"),HI
130 POKE DEC("0C04"),VST:POKE DEC ("0C05"),
    VND
140 SYS DEC("1300")
```

To control the shape, size, and location of the rectangle drawn by these two programs, all you have to do is change the values of the variables in lines 30 through 60 of the RECTANGLE.BAS program. The variables HST and HND set the starting and ending points of the horizontal lines used to draw the rectangle. The starting and ending points of the rectangle's sides are determined by the variables VST and VND. So, by changing the values of these four variables, you can choose the shape, size, and location of the rectangle displayed on the C-128's 320-by-200 dot high-resolution screen.

There are two main reasons why RECTANGLE.S runs so much faster than the SQUARE.S program. First, it doesn't have to perform as many calculations each time it plots a dot on the screen. Second, a number of the major calculations that it does perform are shorter and faster.

Generating a Bit-Mapped Display

To understand what has made these improvements possible, it's necessary to understand how the Commodore 128 produces high-resolution screen graphics in 40-column mode. So here's a brief review of some facts that appeared in chapter 12, together with some new information that may provide you with a better understanding of the program.

Each dot on the C-128's high-resolution screen reflects the state of one bit stored in a screen map that resides in RAM. If the data bit that controls a screen dot is set to 1, then its corresponding dot on the screen is lit. But if the same bit is cleared to 0, then its corresponding dot is dark.

The bit map used to generate the screen in the RECTANGLE.S program begins at memory address $2000, or 8192 in decimal notation. It is labeled SCRBAS (for "screen base") in line 19, and is referred to by that label throughout the program.

How Bit Mapping Works

There are 64,000 dots on a 40-column high-resolution screen, so it takes 64,000 bits—or 8,000 bytes—of memory to store a screenful of bit-mapped data. But, as you may remember from chapter 12, the order in which these 8,000 bytes are stored in memory is very different from the order in which they are displayed on the screen.

In memory, the bytes used to create a screen map are stored in consecutive order, beginning with byte 0 and ending with byte 7,999. But when the C-128 generates a video display, it divides the screen into a grid of 1,000 rectangles, each containing 8 bytes. The 8 bytes that make up each rectangle are stacked one on top of the other. The 1,000 8-byte rectangles on the screen are arranged into a matrix that is 40 columns wide by 25 columns high—the same arrangement that the C-128 uses to generate a 40-column text display.

This kind of screen layout makes it very easy to program a text display, because the 8-byte rectangle that forms each character displayed on the screen in text mode can be fetched from 8 consecutive bytes in memory. But when the C-128 is in 40-column high-resolution mode, bit mapping a dot on a screen becomes considerably more complicated. To plot a dot on a high-resolution screen, a program must carry out three separate operations. First, it locates the 8-byte rectangle in which the dot appears. Then, it determines the byte (or line) inside that rectangle. Finally, it pinpoints the dot's position in that *byte.* Only then can the dot be plotted on the screen.

Before any of these operations can be carried out on a given dot, however, the dot's exact position on the screen must be determined. Because there are 40 columns of rectangles on the screen and 8 horizontal dots in each column, there are 320 horizontal positions in which a dot can appear. The horizontal position of each dot on the screen can be determined by using a set of 320 horizontal coordinates, or X coordinates, which are numbered from 0 to 319. Down the screen, there are 25 rows of 8-byte rectangles. The vertical position of each dot on the screen can be determined by using a set of 200 vertical coordinates, or Y coordinates, which are usually numbered from 0 to 199.

Dot-Plotting Formulas

The first step in converting a dot's screen location into its corresponding bit in memory is to divide the dot's vertical coordinate, or Y coordinate, by 8. The result of this operation is the row number of the 8-byte rectangle in which the dot appears. Then, the dot's horizontal position, or X coordinate, must also be divided by 8. This gives us the column number of the 8-byte rectangle in which the dot appears. Next, the dot's horizontal position within its 8-byte rectangle must be calculated. We can then use the following formula to bring all of the previous formulas together and calculate the screen column of the byte in which the desired dot appears:

BYTE = ROW × 320 + COL × 8 + LINE + BASE

As mentioned previously, the variable ROW in this formula represents the horizontal row in the rectangle that contains the dot, and the variable COL represents the vertical column in the rectangle. BASE represents the starting address of the screen map being used, and LINE represents the line number of the desired byte in a 200-line high-resolution screen (with the lines numbered 0 through 199). The ROW variable is multiplied by 320 because there are 320 dots in a screen line, and the COL variable is multiplied by 8 because there are 8 lines of bytes in each screen rectangle.

After we carry out this last calculation, we must still take care of one complicating factor. The 8 bits of data that form each byte on the screen are in a different order in RAM from the order in which they appear on the screen. In memory, the bits that make up a byte are arranged from right to left. But on the screen, the 8 dots that make up a byte are arranged in the opposite order: from left to right.

We must use one last formula to reverse the positions of the bits in the byte so that they appear in the proper order when they are displayed on the screen. This formula is:

BIT = 7 − (X AND 7)

Y Lookup Table

Now that we know how a dot's position on a screen can be converted into its corresponding position in RAM, we're ready to see exactly how the RECTANGLE.S program differs from the SQUARE.S program. The most important difference is this: Every time the SQUARE.S program plots a dot, it uses the series of formulas just presented to calculate the dot's position on the screen. But RECTANGLE.S does not perform every one of these calculations every time it plots a dot; instead, each time RECTANGLE.S has to plot the position of a dot, it consults a Y lookup table and simply looks up the RAM starting address of the screen line on which the dot appears. The program then calculates the dot's horizontal coordinate, or X offset, and adds it to the Y coordinate address it has found in its Y lookup table. The result of this calculation is the dot's address in RAM. This procedure reduces considerably the number of calculations that must be carried out to plot a dot on a screen and can significantly increase the operating speed of the program.

The Y lookup table used in the RECTANGLE.S program is set up in lines 97 through 141. As the table is created, it is stored in a block of memory that begins at memory address $8000. Actually, two tables are set up in this section of the program; the low byte of each Y address is stored in a table that starts at memory address $8000, and the high byte of each Y address is stored in a second table that begins at $8100. This may sound like a strange way to set up an address table, but it makes good sense because the same offset used to fetch the high byte of a Y address can also be used to fetch the low byte.

Now let's take a closer look at how a Y lookup table works. First, the 8502 Y register is used to create a loop in which the starting address of each line on the screen is calculated. During the loop, the number of each horizontal line on the screen is loaded into the accumulator, beginning with line 0 and ending with line 199.

In lines 106 through 110, each line number is divided by 8 to pinpoint the row of 8-byte rectangles in which the dot appears. But this division is carried out in a streamlined way, not in the slow old-fashioned way that was used in the SQUARE.S program in chapter 12. Each time a line number is loaded into the accumulator, each bit of the number is moved three places to the right using three LSR (logical shift right) instructions. Because the bits in a binary byte progress from right to left in powers of 2, the easiest way to divide a bit by 2 is to shift each bit in the byte one place to the right. Shifting each bit two places to the right is equivalent to dividing the bit by 4, a three-bit shift to the right is the same as dividing by 8, and so on. So three shifts to the right are used to divide the contents of the accumulator by 8.

In lines 112 through 126, the row number that has just been calculated is multiplied by 320 using a multiplication subroutine, which appears in lines 77 through 95. This routine looks similar to the 16-bit multiplication subroutine in the SQUARE.S program, but a close comparison will show that it's a few bytes shorter. And every little bit (or byte) helps when you're trying to speed up a program.

After each row number is multiplied by 320, the product is added to the starting address of the screen map, and the sum is stored in the low byte and high byte lookup tables that start at $8000 and $8100. This procedure continues until both tables are filled in.

After the program creates its Y lookup table, it moves to the process of drawing a rectangle on the screen—with the help of values poked in during the execution of the RECTANGLE.BAS program. When the program has the necessary values, it first calculates the X offset used to display each dot. The program performs this calculation in much the same way that the SQUARE.S program did. Then, in lines 293 through 300, it looks up the starting address of each screen line. Finally, it adds each Y offset address to the appropriate X coordinate with the help of indirect (Y register) addressing, and thus determines the location of the byte in which each dot appears.

Another tricky shortcut is used in lines 308 through 310 of the RECTANGLE.S program. In these lines, the equation:

BIT = 7 − (X AND 7)

is solved by using another table—a very short one that appears in line 52. This formula reverses the order of the bits in a byte before displaying the byte on the screen. We can speed up solving the equation by using a table instead of calculations.

Now you know how to draw lines and rectangles on a high-resolution screen at speeds matching those achieved in commercial graphics programs. In the next chapter, we'll reveal some more tricks of the trade and see how joysticks, paddles, and mice can be used to control fast-action graphics on a high-resolution screen.

14

The Fastest Draw

A high-resolution sketching program for the C-128

Take one joystick-handling program, one high-resolution graphics program, and a little imagination, and what have you got? When I tried that recipe, I came up with the program in this chapter—a high-resolution drawing program for the Commodore 128.

This new program, titled JOYSTICK.S, combines the best features of the JOYSTICK.BAS program in chapter 11 and the RECTANGLE.S program in chapter 13. It's a computerized version of those plastic, carbon-filled sketching screens that you may remember from your childhood. When you type, assemble, and run the program, you'll see what I mean.

JOYSTICK.S Program

The JOYSTICK.S program, in listing 14-1, is a high-resolution version of the JOYSTICK.BAS program presented in chapter 11. But it runs much faster, and it works in a slightly different way. It allows the user to draw on the screen using a game controller, as in the JOYSTICK.BAS program. But it has a slightly different method of reading the controller's trigger button. In the JOYSTICK.BAS program, the trigger button prevents the printing of a dot on the screen. In the JOYSTICK.S program, pressing the joystick trigger completely erases the screen display.

Listing 14-1
JOYSTICK.S
program

```
*
* JOYSTICK.S
*
          ORG    $1300
*
COLOR     EQU    $10
BASE      EQU    $2000
SCROLY    EQU    $D011
BMPTR     EQU    $A2D
BMFLG     EQU    $D8
COLMAP    EQU    $1C00
CIAPRA    EQU    $DC00
*
HMAX      EQU    320
VMAX      EQU    200
HMID      EQU    160
VMID      EQU    100
*
SCRLEN    EQU    8000
MAPLEN    EQU    1000
*
TEMP      EQU    $FA
BYTE      EQU    TEMP+2
*
TABPTR    EQU    TEMP
```

Listing 14-1 cont.

```
TABSIZ    EQU   $9000
*
HPSN      EQU   TABSIZ+2
VPSN      EQU   HPSN+2
CHAR      EQU   VPSN+1
ROW       EQU   CHAR+1
LINE      EQU   ROW+1
BITT      EQU   LINE+2
*
MPRL      EQU   BITT+1
MPRH      EQU   MPRL+1
MPDL      EQU   MPRH+1
MPDH      EQU   MPDL+1
PRODL     EQU   MPDH+1
PRODH     EQU   PRODL+1
*
FILVAL    EQU   PRODH+1
HPOS      EQU   FILVAL+1
JSV       EQU   HPOS+2
STATUS    EQU   JSV+1
NEWVAL    EQU   STATUS+1
*
          JMP   START
*
* BLOCK FILL ROUTINE
*
BLKFIL    LDA   FILVAL
          LDX   TABSIZ+1
          BEQ   PARTPG
          LDY   #0
FULLPG    STA   (TABPTR),Y
          INY
          BNE   FULLPG
          INC   TABPTR+1
          DEX
          BNE   FULLPG
PARTPG    LDX   TABSIZ
          BEQ   FINI
          LDY   #0
PARTLP    STA   (TABPTR),Y
          INY
          DEX
          BNE   PARTLP
FINI      RTS
*
* 16-BIT MULTIPLICATION
*
MULT16   LDA   #0
          STA   PRODL
```

Listing 14-1 cont.

```
          STA   PRODH
          LDX   #17
          CLC
MULT      ROR   PRODH
          ROR   PRODL
          ROR   MPRH
          ROR   MPRL
          BCC   CTDOWN
          CLC
          LDA   MPDL
          ADC   PRODL
          STA   PRODL
          LDA   MPDH
          ADC   PRODH
          STA   PRODH
CTDOWN    DEX
          BNE   MULT
          RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8
*
PLOT      LDA   VPSN
          LSR   A
          LSR   A
          LSR   A
          STA   ROW
*
* CHAR=HPSN/8
*
          LDA   HPSN
          STA   TEMP
          LDA   HPSN+1
          STA   TEMP+1
          LDX   #3
DLOOP     LSR   TEMP+1
          ROR   TEMP
          DEX
          BNE   DLOOP
          LDA   TEMP
          STA   CHAR
*
* LINE=VPSN AND 7
*
          LDA   VPSN
          AND   #7
          STA   LINE
*
```

Listing 14-1 cont.

```
* BITT=7-(HPSN AND 7)
*
          LDA   HPSN
          AND   #7
          STA   BITT
          SEC
          LDA   #7
          SBC   BITT
          STA   BITT
*
* MULTIPLY ROW * HMAX
*
          LDA   ROW
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   #<HMAX
          STA   MPDL
          LDA   #>HMAX
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
          STA   TEMP
          LDA   MPRL+1
          STA   TEMP+1
*
* ADD PRODUCT TO BASE
*
          CLC
          LDA   #<BASE
          ADC   TEMP
          STA   TEMP
          LDA   #>BASE
          ADC   TEMP+1
          STA   TEMP+1
*
* MULTIPLY 8 * CHAR
*
          LDA   #8
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   CHAR
          STA   MPDL
          LDA   #0
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
          STA   BYTE
```

Listing 14-1 cont.

```
          LDA    MPRH
          STA    BYTE+1
*
* ADD LINE
*
          CLC
          LDA    BYTE
          ADC    LINE
          STA    BYTE
          LDA    BYTE+1
          ADC    #0
          STA    BYTE+1
*
* BYTE = BYTE + TEMP
*
          CLC
          LDA    TEMP
          ADC    BYTE
          STA    BYTE
          LDA    TEMP+1
          ADC    BYTE+1
          STA    BYTE+1
*
* BYTE=BYTE OR 2↑BIT
*
          LDX    BITT
          INX
          LDA    #0
          SEC
SQUARE    ROL
          DEX
          BNE    SQUARE
          STA    NEWVAL
          RTS
*
* MAIN ROUTINE
*
* DEFINE BIT MAP AND
* ENABLE HI-RES GRAPHICS
*
START     STA    $FF01
          LDA    #$78
          STA    BMPTR
*
          LDA    #$20
          STA    BMFLG
*
          LDA    #0
          STA    $FF00
```

Listing 14-1 cont.

```
         LDA   SCROLY
         ORA   #$20
         STA   SCROLY
         STA   $FF01
*
* CLEAR BIT MAP
*
         LDA   #0
         STA   FILVAL
         LDA   #<BASE
         STA   TABPTR
         LDA   #>BASE
         STA   TABPTR+1
         LDA   #<SCRLEN
         STA   TABSIZ
         LDA   #>SCRLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* SET BKG AND LINE COLORS
*
         LDA   #COLOR
         STA   FILVAL
         LDA   #<COLMAP
         STA   TABPTR
         LDA   #>COLMAP
         STA   TABPTR+1
         LDA   #<MAPLEN
         STA   TABSIZ
         LDA   #>MAPLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* PRINT DOT AT MIDSCREEN
*
         LDA   #VMID
         STA   VPSN
         LDA   #<HMID
         STA   HPSN
         LDA   #>HMID
         STA   HPSN+1
         JSR   PRINT
*
* READ JOYSTICK
*
* FIRST CHECK TRIGGER BUTTON
*
READJS   LDA   #0
         STA   $FF00      ;BANK 15
```

Listing 14-1 cont.

```
         LDA   CIAPRA
         STA   STATUS
         STA   $FF01       ;BANK 0
         AND   #$10
         BEQ   START
*
* NOW READ JOYSTICK
*
         LDA   #$0F
         PHA
         AND   STATUS
         STA   JSV
         PLA
         SEC
         SBC   JSV
         STA   JSV
*
         TAX
         BEQ   READJS
         LDA   RELADS-1,X
         STA   MODREL+1
MODREL   BNE   *
MODR1
*
* ROUTINES TO MOVE JOYSTICK
*
UP       JSR   MOVEUP
         JSR   PRINT
         JMP   READJS
*
DOWN     JSR   MOVEDN
         JSR   PRINT
         JMP   READJS
*
DNANDL   JSR   MOVEDN
         JMP   LEFT
UPANDL   JSR   MOVEUP
LEFT     LDX   HPSN
         LDY   HPSN+1
         TXA
         BNE   DECLSB
         DEY
DECLSB   DEX
         STX   HPSN
         STY   HPSN+1
         JSR   PRINT
         JMP   READJS
*
DNANDR   JSR   MOVEDN
         JMP   RIGHT
```

Listing 14-1 cont.

```
UPANDR   JSR   MOVEUP
RIGHT    LDX   HPSN
         LDY   HPSN+1
         INX
         BNE   NOINC
         INY
NOINC    STX   HPSN
         STY   HPSN+1
         JSR   PRINT
         JMP   READJS
*
* SUBROUTINES TO MOVE UP & DOWN
*
MOVEUP   LDX   VPSN
         DEX
         STX   VPSN
         RTS
*
MOVEDN   LDX   VPSN
         INX
         STX   VPSN
         RTS
*
* MAKE SURE DOT IS WITHIN RANGE
*
CHECK    LDA   VPSN
         BEQ   RAISE
         CMP   #VMAX-1
         BCS   LOWER
         JMP   HCHECK
RAISE    INC   VPSN
         JMP   HCHECK
LOWER    LDA   #VMAX-1
         STA   VPSN
*
HCHECK   BIT   HPSN+1
         BPL   OKLOW
         LDA   #1
         STA   HPSN
         LDA   #0
         STA   HPSN+1
         RTS
*
OKLOW    LDA   #<HMAX-2
         CMP   HPSN
         LDA   #>HMAX-2
         SBC   HPSN+1
         BCC  TOOHI
         RTS
*
```

Listing 14-1 cont.

```
TOOHI    LDA   #<HMAX-2
         STA   HPSN
         LDA   #>HMAX-2
         STA   HPSN+1
         RTS
*
* PRINT DOT ON SCREEN
*
PRINT    JSR   CHECK
         JSR   PLOT
         JSR   DRAWDOT
*
         LDA   HPSN
         PHA
         LDA   HPSN+1
         PHA
*
         LDA   HPSN
         BNE   SKIP
         DEC   HPSN+1
SKIP     DEC   HPSN
         JSR   CHECK
*
         JSR   PLOT
         JSR   DRAWDOT
*
         PLA
         STA   HPSN+1
         PLA
         STA   HPSN
         RTS
*
DRAWDOT  LDY   #0
         ORA   (BYTE),Y
         STA   (BYTE),Y
         RTS
*
RELADS   DFB   UP-MODR1
         DFB   DOWN-MODR1
         DFB   READJS-MODR1
         DFB   LEFT-MODR1
         DFB   UPANDL-MODR1
         DFB   DNANDL-MODR1
         DFB   READJS-MODR1
         DFB   RIGHT-MODR1
         DFB   UPANDR-MODR1
         DFB   DNANDR-MODR1
*
```

If you understand how the RECTANGLE.S and JOYSTICK.BAS programs work, you probably won't have any trouble understanding the basic principles of the JOYSTICK.S program. It plots on the screen with the same techniques used in RECTANGLE.S and it reads joysticks with techniques similar to the ones in JOYSTICK.BAS. But there are also some new features.

One of these features is an error-checking subroutine in lines 344 through 376. This subroutine is quite straightforward, but it's also quite important because it prevents what's printed on the screen from extending beyond the boundaries of RAM space designated as screen memory. Error-checking routines such as this one are very important in assembly language programming because they keep screen data from overshooting their boundaries and ending up in memory blocks where they shouldn't be. When data goes bounding into "never-never land," it can bring a program to a crashing halt. And you don't want that to happen in your assembly language programs.

Another important feature of the JOYSTICK.S program is a short routine that appears in lines 290 through 300. This segment of code is a relative address modification routine, and it takes advantage of a strange and wonderful capability of assembly language and other programming languages: the ability of a program to modify itself.

The relative address modification routine in the JOYSTICK.S program serves the same purpose as an ON...GOTO routine in BASIC. It reads a numeric value—in this case, a value provided by a joystick—and branches to a routine that has been assigned a corresponding value in a program. This is a fairly sophisticated programming technique, even in BASIC. So, before we examine how it works in assembly language, let's take a brief look at a somewhat simpler type of address modification program. The short segment of code in listing 14-2 is not actually a *relative* address modification routine, like the one in the JOYSTICK.S program, but rather a *direct* address modification subroutine that is used more often in assembly language programs. When you understand the principle of direct address modification, it's easier to grasp the relative address modification technique used in the JOYSTICK.S program.

Listing 14-2
Simple address modification routine

```
Memory                     Line          Source Code Listing
Address  Object Code  Number      Label              Code

8040     AD A7 02      100      ADDRESS  LDA  VALUE
8043     EE 41 80      101               INC  ADDRESS+1
8046     D0 03         102               BNE  NEXT
8048     EE 42 80      103               INC  ADDRESS+2
804B     60            104      NEXT     RTS
```

The segment of code in listing 14-2 shows both the source code and the object code of a short address modification routine, in addition to the addresses of the memory registers into which the object code is stored. You can get a clearer picture of how the program works by looking at its object code: the machine language part of the listing.

Look carefully at the routine's object code, and you'll see that when the subroutine is first called, the accumulator is loaded with a value labeled—logically enough—VALUE. As you can see in the object code listing of line 100, that value is fetched from memory register $02A7.

In the next three lines of the routine, something quite extraordinary happens. As you may be able to recognize by now, the instructions in lines 101 through 103 are a standard set of instructions for incrementing a 16-bit number. But what number is incremented here? Well, look again at the object code part of the program, and you'll see that the value that is incremented is whatever 16-bit value is stored in memory registers $8041 and $8042. What value is that? Why, it's the value that follows the LDA mnemonic in line 100.

Now take a very close look at the object code listing of this routine, and you'll see that the routine has now rewritten itself! The next time the routine is called, line 100 loads the accumulator not with the value stored in memory register $02A7 but with that value plus one—and that value continues to be incremented by one every time the routine is called!

Address modification is a very powerful programming technique used quite often in high-performance assembly language programs. Routines that use address modification are compact and run fast, and they do not require the use of page zero memory, which is always in short supply. So a good knowledge of the principles of address modification can be of great value to the assembly language programmer.

Relative Address Modification

Now let's take a look at relative address modification: the kind used in the JOYSTICK.S program. As already mentioned, assembly language programs use relative address modification in much the same way that BASIC programs use ON...GOTO routines. In JOYSTICK.S, relative address modification makes the program branch to an UP, DOWN, LEFT, or RIGHT routine—or a combination of these—depending on the direction that the joystick is moved.

The address modification routine in JOYSTICK.S uses a data table at the end of the program, in lines 408 through 417. As you can see, this data table is labeled RELADS, which stands for "relative address." But notice that the values of the bytes in the RELADS table are not defined as specific values. Instead, each value in the table is defined as the result of a calculation: specifically, the difference between the value in the table and a given line in the JOYSTICK.S program.

Look carefully at the definitions of the bytes in the RELADS table, and you'll see that each value in the table has been defined as being equal to the address of one specific joystick movement routine, minus the value of the address of line 292 of the JOYSTICK.S program, which is labeled MODREL (an abbreviation, in a backward sort of way, for "relative modification.")

Now examine lines 290 through 300, and you'll see how an assembly language program can rewrite itself using the technique of relative address modification. In line 290, the beginning of the address modification routine, the direction switch of the game controller has just been read, and the value obtained is stored in the 8502 chip's X register. If the controller's trigger button is currently being pressed, the screen is cleared and the joystick is read again. But if the trigger button has not been pressed, the accumulator is loaded with an 8-bit value that points to a specific address: the address of one of the joystick movement routines in lines 301 through 342.

Next, examine lines 290 and 291. In line 290, an 8-bit value pointing to the desired address is loaded into the accumulator. Then, in line 291, that value is stored in a given memory address. How is that memory address obtained? It has to be calculated using the label/offset combination MODREL+1.

Where is MODREL+1? The answer is in line 292:

```
292 MODREL   BNE   *
```

The label of that line, as you can see, is MODREL. From a machine language point of view, you can also say that MODREL is the label of one specific memory register: the register that holds the machine language equivalent of the assembly language mnemonic BNE. So, if you want to assign a label to the address of the BNE mnemonic in line 292 of the JOYSTICK.S program, you have to use the label MODREL.

If the BNE mnemonic is located at the address labeled MODREL, then what's at the address labeled MODREL+1? Well, in the source code listing of the JOYSTICK.S program, MODREL+1 appears to be the address of an asterisk. This may look like a strange way to write a line of code—and it is. But in most assembly language programs (including those written using the Merlin 128 assembler), an asterisk does have a meaning. It's a pseudo op that is often used to refer to the current contents of an assembler's program counter. When the JOYSTICK.S program is first assembled, the 16-bit value stored in memory registers MODREL+1 and MODREL+2 is nothing but a pointer to its own starting address!

When the JOYSTICK.S program is executed, however, the contents of MODREL+1 and MODREL+2 change automatically. In lines 290 and 291, the contents of MODREL+1 is changed to the value stored in the accumulator—which is, in turn, the value of a specific byte in the data table labeled RELADS. As we have seen, each byte in that table is an 8-bit pointer that can be used to calculate the address of a specific joystick movement routine.

Relative address modification is a sophisticated concept, so don't be surprised if it all seems a little foggy at first. In summary, when the JOYSTICK.S program reaches lines 290 and 291, a value pointing to the address of a joystick movement routine is placed in an address designated as MODREL+1. When that happens, the value of the asterisk in line 292 is replaced with a 16-bit value pointing to the address of a specific byte in the RELADS table in lines 408 through 417. This byte is then used to calculate the final address of the desired joystick movement routine.

If you don't quite understand all of this yet, don't worry; just type, assemble, and run the JOYSTICK.S program, and observe what the address modification routine actually does. When you understand what it does, then understanding how it does it will be less of a problem.

You have seen only two kinds of address modification routines in this chapter, but many other kinds of address modification and data modification techniques are used in assembly language. Some assembly language programmers don't like to use self-modifying code because it tends to be difficult to decipher after the ink is dry, and because it isn't compatible with the modular style of program design that has become popular over the past few years. But self-modifying code is still widely used, especially in programs written for small computers, because it's compact, fast running, and doesn't require the use of page zero. In programs that require frequent use of index registers, self-modifying code can also take some of the workload off the 8502's X and Y registers, leaving them free for other duties.

15

Commodore 128 Music and Sound

And an introduction to interrupt operations

One of the best features of the Commodore 128 is its ability to synthesize music and sounds. Despite its user-friendly price, the C-128 has sound-generating and music-generating capabilities that rival those of music synthesizers used by professional musicians. In this chapter, you'll learn how to turn your C-128's keyboard into a music synthesizer keyboard that can produce an almost limitless variety of sounds.

The C-128's built-in synthesizer can be programmed in either BASIC or assembly language. But assembly language is a much better choice—for many reasons. Here are a few:

- To write musical programs for the C-128, you need to manipulate bits in memory registers—a job that is slow and clumsy in BASIC, but fast and easy in assembly language.
- Timing is often critical in sound and music programming, so the speed of assembly language is especially important in programs that deal with music and sound.
- The length of a note cannot be determined very precisely in BASIC, but musical timing can be controlled with pinpoint precision in assembly language. In fact, by using a programming tool called an interrupt, you can make the lengths of musical notes, rests, and phrases independent of everything else in an assembly language program. By using interrupts, you can add music and sound to an assembly language program with perfect synchronization, and you can be certain that your soundtrack will always run at the same speed, no matter how many other features are then added to the program. You'll learn how to use interrupts in this chapter.

To understand how the Commodore 64/128 music synthesizer works, it helps to have at least a passing familiarity with the science of sound. You don't have to be a musician or an audio engineer to write sound programs for the C-128, but it's good to know a little bit about how a music synthesizer produces sound.

When you hear a sound from a musical instrument, there are four characteristics that combine to create the sound you perceive. These four characteristics are:

- Volume, or loudness
- Frequency, or pitch
- Timbre, or sound quality
- Dynamic range, or the difference in level between the loudest sound that can be heard and the softest sound that can be heard during a period of time

SID Chip

In the Commodore 128, there is a special microprocessor that can be programmed to control the volume, frequency, timbre, and dynamic range of sounds. This processor, called the 6581 SID (Sound Interface Device), gives the Commodore 128 its outstanding sound-synthesizing capabilities.

The SID chip has three separate voices, and each of these voices can be programmed independently. This means that the C-128 can play music in three-part harmony—or, if you prefer, you can use one voice for melody, one for percussion, and one for bass. You can also use the SID chip to generate noises instead of music, and you can program each of SID's three voices to produce a different sound. SID can even be taught to mimic human speech, but that requires sophisticated programming.

Shortly, you'll get a chance to see—and hear—some of the things that SID can do. But first we'll have to learn some basic facts about the SID chip, such as its location in the C-128's RAM and how it can be accessed in assembly language programs.

Where SID Lives

In the Commodore 128, 29 memory registers—$D400 through $D41C —are used to address the SID chip. Table 15-1 shows how these 29 registers are used in sound and music programming.

Table 15-1 Memory Blocks Used by the 6581 SID Chip

Memory Block	Contents of Block
$D400 through $D406	Registers for voice 1
$D407 through $D40D	Registers for voice 2
$D40E through $D414	Registers for voice 3
$D415 through $D418	Sound filter and volume controls
$D419 through $D41A	Game paddle registers (not used for sound)
$D41B	Oscillator 3 read byte and random-number generator (used to modulate output of other registers)
$D41C	Voice 3 envelope generator read byte (used to modulate output of other registers)

SID's Memory Map

Table 5-2 is a memory map of the SID chip's sound-related registers, arranged by the voices and other functions affected by each register. The functions of most of the registers listed in this table are explained later in this chapter. Functions not covered can be found in various Commodore 128 reference manuals, such as the *Commodore 128 Reference Guide for Programmers* and the *Commodore 128 Programmer's Reference Guide.*

As tables 15-1 and 15-2 show, registers $D400 through $D418 are the only SID registers ordinarily used in basic-level to intermediate-level SID programming. The largest block of memory in the tables, the section that extends from $D400 through $D414, can be divided into three subsections: one for voice 1, one for voice 2, and

Table 15-2 Memory Map of the 6581 SID Chip Registers

Address	Label	Function
$D400	FRELO1	Voice 1 frequency control (low byte)
$D401	FREHI1	Voice 1 frequency control (high byte)
$D402	PWLO1	Voice 1 pulse waveform width (low byte)
$D403	PWHI1	Voice 1 pulse waveform width (high nibble)
$D404	VCREG1	Voice 1 control register
$D405	ATDCY1	Voice 1 attack/decay register
$D406	SUREL1	Voice 1 sustain/release control register
$D407	FRELO2	Voice 2 frequency control (low byte)
$D408	FREHI2	Voice 2 frequency control (high byte)
$D409	PWLO2	Voice 2 pulse waveform width (low byte)
$D40A	PWHI2	Voice 2 pulse waveform width (high nibble)
$D40B	VCREG2	Voice 2 control register
$D40C	ATDCY2	Voice 2 attack/decay register
$D40D	SUREL2	Voice 2 sustain/release control register
$D40E	FRELO3	Voice 3 frequency control (low byte)
$D40F	FREHI3	Voice 3 frequency control (high byte)
$D410	PWLO3	Voice 3 pulse waveform width (low byte)
$D411	PWHI3	Voice 3 pulse waveform width (high nibble)
$D412	VCREG3	Voice 3 control register
$D413	ATDCY3	Voice 3 attack/decay register
$D414	SUREL3	Voice 3 sustain/release control register
$D415	CUTLO	Filter cutoff frequency (low nibble)
$D416	CUTHI	Filter cutoff frequency (high byte)
$D417	RESON	Filter resonance control register
$D418	SIGVOL	Volume and filter select register

one for voice 3. Later in this chapter, the functions of all of the registers in the block that extends from $D400 to $D414 are covered in more detail.

SID's Functions

Let's take an overall look at how the SID chip's registers are used to program the volume, frequency, timbre, and dynamic range of the three voices of the Commodore 128.

Volume

For some reason, the designers of the Commodore 128 made it impossible to control the volume of the SID chip's three voices individually. Instead, the loudness of the overall sound produced by the SID register is determined by the value placed in the lower four bits (bits 0 through 3) of memory register $D418. This register is sometimes known as the SIGVOL register.

To control the volume of all sounds produced by the SID chip, just place a value ranging from $0 to $F in the lower nibble of the SIGVOL register. The larger the value of this nibble, the louder the sound that the SID chip produces. If the value of the nibble is $0, no sound is generated. In most applications, the volume nibble of the SIGVOL register is kept at $F, its maximum setting.

Bits 4 through 6 of the SIGVOL register control three sound filters built into the SID chip: a low-pass filter, a bandpass filter, and a high-pass filter. The uses of these filters are explained later in this chapter.

By setting bit 7 of the SIGVOL register to 1, you can disconnect the output of the SID chip's voice 3. When voice 3 is disconnected, the oscillator that voice 3 is equipped with can be used for modulating the sound of the other two voices. The voice 3 oscillator can also be used for other purposes—such as generating random numbers—without affecting the output of sound.

When the filters controlled by register $D418 are not being used and when there is no need to disconnect voice 3, the SID chip's volume can be controlled by simply storing a value ranging from $0 to $F (or from 1 to 15 in decimal notation) in the SIGVOL register. But when bits 5 through 7 of the SIGVOL register are in use, you must use masking operations to implement a desired volume setting without affecting the register's other functions. Listing 15-1 is a segment of code that could be used to implement a volume setting of 15 ($F in hexadecimal notation) without disturbing the high-order nibble of the SIGVOL register.

Listing 15-1
Masking operation for setting volume

```
LDA SIGVOL
AND #$F0
ORA #$0F
STA SIGVOL
```

Frequency

The pitch of a musical note is determined by its frequency. Frequency is usually measured in hertz (Hz), or cycles per second. The frequencies that can be produced by the Commodore 128's SID chip range from 0 Hz (very low) to 4,000 Hz (quite high).

The SID chip synthesizes the frequencies of sounds by carrying out a rather complex mathematical operation. First, it reads a pair of 8-bit values (one "low" value and one "high" value) in a pair of frequency control registers. The SID chip has six such registers—two for each voice—and the addresses of all of them are listed in table 15-1.

When a pair of frequency control registers are loaded with two 8-bit values, the SID chip combines them into a 16-bit value. It then divides that 16-bit value by a number derived from a certain frequency: specifically, the frequency of a system clock built into the Commodore 128. Then the SID chip can generate a note of the desired frequency.

That's quite an involved series of operations, but you don't have to worry about how they all work to produce a note of a given frequency on the Commodore 128. All you have to do is place the proper values in the proper memory registers, and then set a certain bit in another register. All of the values you need to play eight octaves of notes on the Commodore 128 are listed in appendix E. In this appendix, there are two values (a "low" value and a "high" value) that must be placed in the SID chip's frequency control registers to produce each note that the Commodore 128 can generate. But remember that the values listed in this table are not actual frequencies; they are numbers the SID chip uses to calculate frequencies.

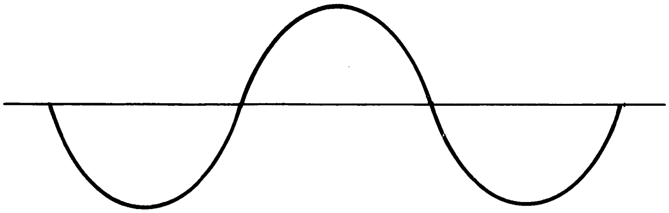
Timbre

Timbre, or note quality, can be illustrated with the help of a structure called a waveform. The SID chip can generate four kinds of waves: a triangle wave, a sawtooth wave, a pulse wave, and a noise wave. To understand the concept of waveforms, you need to know a little about harmonics. So here is a crash course in music theory.

With an electronic instrument, it is possible to generate a tone that has only one pure frequency. But when a note is played on a musical instrument, more than one frequency is usually produced. In addition to a primary frequency, or a fundamental, there is usually a set of secondary frequencies called harmonics. It is this total harmonic structure that determines the timbre of a sound.

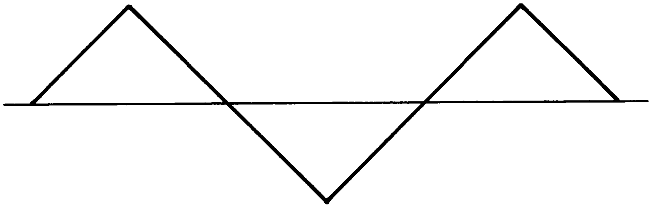
When a tone containing only a fundamental frequency is viewed on an oscilloscope, the pattern produced on the screen is a pure sine wave, as shown in figure 15-1. When a flute is played, the waveform it produces is very close to that of a pure sine wave.

Figure 15-1
Sine waveform



When harmonics are added to a tone, the result is a richer sound, which produces a triangle wave. The waveform of a triangle wave is shown in figure 15-2. Triangle waveforms, or waves that are close to triangle waveforms, are produced by instruments such as xylophones, organs, and accordians.

Figure 15-2
Triangle waveform



When still more harmonics are added to a note, other kinds of waves are formed. Harpsichords and trumpets, for example, produce a type of wave called a sawtooth wave, which is shown in figure 15-3.

A piano generates a squarish kind of wave called a square wave or a pulse wave, as shown in figure 15-4.

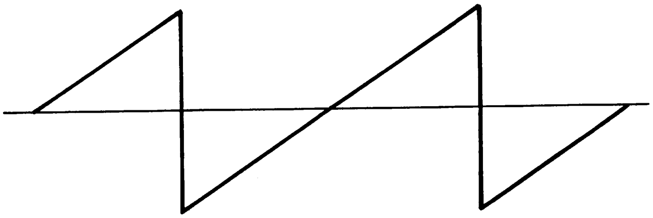


Figure 15-3 Sawtooth waveform

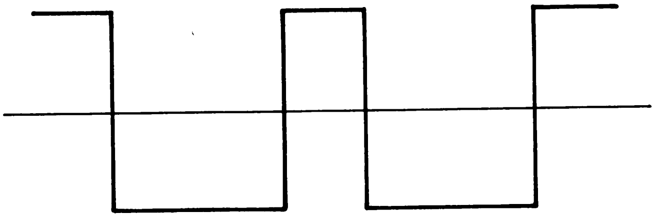


Figure 15-4 Pulse waveform

Pulse Waveform Width Control

When the SID chip is called on to generate a pulse wave, you need to use an additional control called a pulse waveform width control. As you can see in figure 15-4, the pulses in a pulse waveform have a certain width, and the gaps separating the pulses may have a different width. The SID chip has six registers—two for each voice—that can be used to control the widths of pulse waveforms. A pulse wave generated by the SID chip has a 12-bit resolution, so only 12 bits in each pair of width control registers are used: all 8 bytes of each low-order register, plus the lower nibble of each high-order register.

The setting of each width control register determines how long a pulse wave will stay at the high part of its cycle. The range of 12-bit values, from 0 to 4,095, makes it possible for a square wave to stay in the high part of its cycle from 0% to 100% of the time, in 4,096 steps.

Another kind of waveform that the SID chip can produce is a noise waveform. A noise waveform creates a random sound output that varies with a frequency proportionate to that of the oscillator built into voice 1. Noise waveforms are often used to imitate the sounds of explosions, drums, and other nonmusical noises.

How to Select a Waveform

The SID chip has three registers—one for each voice—that can be used to determine the waveforms of sounds. These three registers, called control registers, are $D404 (for voice 1), $D40B (for voice 2), and $D412 (for voice 3).

These three registers are multipurpose registers; only their high-order nibbles (bits 4 through 7) are used for determining waveforms.

The uses of the remaining bits are discussed later in this chapter. Meanwhile, these are the bits that must be set to select a waveform:

Bit 4	Triangle waveform
Bit 5	Sawtooth waveform
Bit 6	Pulse waveform
Bit 7	Random noise waveform

Filters

Many other kinds of waves can be produced with the help of special filters. Three such filters—a low-pass filter, a high-pass filter, and a bandpass filter—are built into the Commodore 128. A low-pass filter masks out frequencies above a certain cutoff frequency and attenuates the low frequencies that pass through. A high-pass filter masks out frequencies below a certain cutoff frequency and attenuates the high frequencies that pass through. A bandpass filter cuts off frequencies that are outside a range near the center of the frequency spectrum, and attenuates the midrange frequencies that pass through.

As explained in the section dealing with volume, SID register $D418—the register that controls volume—also controls the SID chip's three sound filters. For the sake of simplicity, the filters built into the SID chip are not used in the program presented in this chapter. But you are encouraged to experiment with the filters when you run the program, because you may want to use these filters in your own programs.

Dynamic Range

The dynamic range of a note is the difference in volume between its loudest sound level and its softest sound level in a given period of time. This period of time can range between the time it takes to play a single note and the length of a much longer listening experience, such as a musical performance or a complete musical recording. Dynamic range can be illustrated in many ways. To illustrate and control the dynamics of notes produced by the SID chip, engineers who designed the Commodore used a device called an ADSR envelope, or attack/decay/sustain/release envelope. An ADSR envelope illustrates four distinct stages in the life of a note: four phases that every note undergoes between the time it starts and the time it fades away. These four phases—called attack, decay, sustain, and release—are shown in the ADSR envelope illustrated in figure 15-5.

The addresses of the SID registers used to create ADSR envelopes are listed in table 15-2. As you can see by looking at this table, the SID chip has six registers—two for each voice—that control the attack, decay, sustain, and release characteristics of notes. Each voice has one register that controls the attack and decay phases of notes, and another register that controls the sustain and release phases of

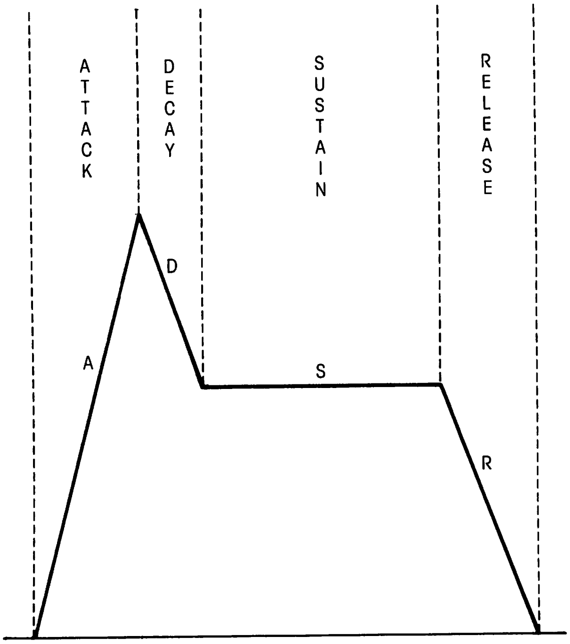


Figure 15-5
ADSR envelope

notes. Following are brief descriptions of the four note cycles identified at the top of figure 15-5.

Phases 1 and 2: Attack and Decay

Every note starts with an attack. The attack phase of a note is the length of time it takes for the volume of the note to rise from a level of zero to the note's peak volume. As soon as a note reaches its peak volume, it begins to decay. The decay phase of a note is the length of time it takes for the note to decay from its peak volume to a predefined sustain volume.

As mentioned earlier, each of the SID chip's three voices has one register that controls both the attack and decay characteristics of notes. The three SID registers that control attacks and decays are $D405 (for voice 1), $D40C (for voice 2), and $D413 (for voice 3). The high nibble of each of these registers (bits 4 through 7) sets the duration of a note's attack cycle, and the low nibble of each register (bits 0 through 3) sets the duration of a note's decay cycle. Each nibble can be set to a value ranging from $0 (for a duration of 2 milliseconds) to $F (for a duration of 8 seconds). The most common settings range somewhere between these two extremes.

Phases 3 and 4: Sustain and Decay

When the decay phase of a note ends, the note is usually sustained for a certain period of time at a certain volume. Then a release phase begins. During this final phase, the volume of the note drops from its sustain level back down to zero.

Each of the SID chip's three voices has one register that controls both the sustain and release characteristics of notes. The three SID registers that control the sustain and release phases of notes are $D406 (for voice 1), $D40D (for voice 2), and $D414 (for voice 3).

The low nibble of each of these registers (bits 0 through 3) sets the duration of a note's release cycle. Each of these "release" nibbles can be set to a value ranging from $0 (for 6 milliseconds) to $F (for 24 seconds). The normal settings of the SID chip's release nibble are somewhere between these two extremes.

The high nibble (bits 4 through 7) of each sustain/release register controls the sustain cycle of a note. But this nibble is not used to control the duration of the sustain cycle. Instead, it is used to control the volume that is maintained throughout the sustain cycle. The duration of a note's sustain cycle must be controlled with either a timing loop or some other kind of timer. The value of the sustain nibble of a sustain/release register can range from $0 (for no volume) to $F (equal to the note's peak volume).

SID's Control Registers

After you've determined a note's volume, frequency, waveform, and dynamic range, it's easy to instruct the SID chip to play the note. All you have to do is set one bit in one register: specifically, the gate bit in the control register for the SID voice that you're using.

The SID chip has three control registers: one for each voice. Their addresses are $D404 (for voice 1), $D40B (for voice 2), and $D412 (for voice 3). These three registers were mentioned earlier in this chapter; their high-order nibbles select the waveforms that the SID chip generates. Now we're ready to talk about their low-order nibbles (bits 0 through 3). The uses of these bits are described in reverse order, beginning with bit 3.

Bit 3, the test bit of each SID control register, disables the oscillator that's built into the voice that the register controls. When this oscillator is disabled, complex waveforms—even waveforms that synthesize speech—can be generated under software control.

Bit 2 of each SID control register is called a ring modulation bit. When this bit is set to 1, the triangle waveform of the voice controlled by this register is replaced with a ring modulated combination of two oscillators, and can thus be used to simulate the sound of a bell or a gong.

Bit 1, a synchronization bit, synchronizes the fundamental frequency of oscillator 1 with the fundamental frequency of oscillator 3, enabling the advanced programmer to create a wide range of complex harmonic structures using voice 1.

Bit 0 is the main bit, or gate bit, of each SID control register. When you've selected a note's volume, frequency, waveform, and dynamic range, and have given the SID chip all the information it needs to play the note, you can start the note by setting the gate bit of the proper SID control register. To stop the note—whether or not it has finished playing—just clear the gate bit of the appropriate SID control register. After you clear the gate bit, you can change the settings of any SID registers. Then you can play another note—or create another sound—by setting the gate bit again. Or, if you prefer, you can play the same note or create the same sound over and over, by repeatedly setting and clearing the gate bit while the other SID registers remain the same.

Using Interrupt Routines

Shortly, you'll have an opportunity to type, assemble and execute an assembly language program that illustrates some of the music-synthesizing capabilities of the SID chip built into your Commodore. First, though, it might be helpful to discuss the concept of the *interrupt,* a very powerful programming technique that is often used in music and sound routines (as well as in many other kinds of high-performance programs).

An interrupt, often cryptically referred to as an IRQ, is a high-priority routine that interrupts other routines so that it can do its work. No matter what is happening when an interrupt is called, a computer will stop everything it is doing to process the interrupt. An interrupt, in other words, always goes to the head of the line and keeps other routines waiting while it does its job.

Assembly language programmers often use interrupts when they want to write time-critical routines. For example, one very important interrupt routine is built into the operating system of the Commodore 128. This routine, called a hardware interrupt routine, takes place 60 times a second, with quartz clockwork precision. During this interrupt, also called a vertical blank interrupt, the screen is blacked out briefly and many vital and time-critical operations take place. For example, the computer's software clock is updated, the keyboard is read, and a cursor-blinking operation is performed. Every 1/60 of a second, when it's time for a hardware interrupt, the interrupt takes place and all other processing is temporarily halted. Not until the interrupt is completed does normal processing resume.

The hardware interrupt routine is very important to the Commodore assembly language programmer because it can be customized with the help of a vector called the hardware interrupt vector. This vector is situated at memory addresses $0314 and $0315. It is often labeled the CINV vector in Commodore 128 programs.

"Stealing" the C-128'S Interrupt Vector

Because the CINV vector is in a documented location in RAM, you can "steal" it any time you like. This means you can make it point to any user-written routine instead of to the hardware interrupt vector that's built into the computer's operating system. Then, 60 times every second, with precise regularity, your own routine will be processed as an interrupt routine.

If you steal the CINV vector in this fashion, however, there are two potential problems you'll have to solve. Here is one: If you want the computer's operating system to continue to work normally, even though its CINV vector has been stolen, you'll have to make sure that all of the operations normally carried out by the hardware interrupt vector still take place.

Fortunately, this is not a difficult task. If you want to make sure that your own interrupt and the normal CINV interrupt both take place every 1/60 of a second, all you have to do is take two simple steps: (1) change the CINV vector to point to your own interrupt, and then (2) end your own interrupt with a jump to the address pointed to by the C-128's original CINV vector.

Here's the second problem that must be solved in writing an interrupt routine. The CINV vector consists of two 8-bit memory registers that, in combination, always hold a 16-bit address. Therefore, when you want to alter the CINV vector, it must be changed in two steps. First, the low byte of the address that the vector points to must be changed. Then the high byte must be changed.

Normally, this sort of operation is no problem to the assembly language programmer. But the CINV vector is a very special sort of vector; its job is to direct the computer to an operation that is carried out 60 times every second. There is always a chance, therefore, that the CINV vector will become active after one of its bytes has been changed but before the other byte has been changed. If that happens, the CINV vector may point to an incorrect address when it is called, resulting in a program crash or a system failure.

SEI and CLI Instructions

To prevent this kind of catastrophe from taking place, the 6502/8502 chip has two special instructions for dealing with interrupts. One of these instructions is SEI, which stands for "set interrupt disable." The other is CLI, which means "clear interrupt disable." When an SEI instruction is invoked during the processing of an assembly language program, the interrupt disable flag of the processor status register is set and no maskable interrupts can take place. (The CINV interrupt is a maskable interrupt.) When a CLI instruction is used during an assembly language program, it has the opposite effect; the interrupt disable flag of the P register is cleared and maskable interrupts are enabled.

Because the SEI and CLI instructions can enable and disable interrupts so easily, they can be used to change the C-128's CINV vector with complete safety. To make sure that a program doesn't crash during the alteration of the CINV vector, just use the SEI instruction before the vector is changed, and then use the CLI instruction after it's changed. When you take that simple precaution, you can be sure that no interrupts take place while the vector is being altered, and that the vector is cleanly and safely changed.

MUSIC Program

In listing 15-2, a program titled MUSIC, the CINV vector is stolen so that a note-timing loop can be inserted into the CINV vector. This ensures that the musical notes produced by the program are always precisely timed. The CINV vector is stolen and altered in lines 57 through 66 of the MUSIC program. In lines 57 through 60, the CINV address ordinarily pointed to by the C-128's built-in CINV vector is stored in a pair of memory registers called USERADD and USERADD+1. Next, in line 61, the SEI instruction disables maskable interrupts. When that has been accomplished, the address of a user-written routine (a note-timing loop) is stored in the address of the CINV vector. Then interrupts are re-enabled with a CLI instruction. The note-timing routine that this operation adds to the C-128's CINV vector is labeled WAIT. It appears in lines 113 through 119 of the MUSIC program.

Listing 15-2
MUSIC program

```
 1 *
 2 * MUSIC
 3 *
 4               ORG    $1300
 5 *
 6 SFDX          EQU    $D4
 7 *
 8 CINV          EQU    $314
 9 USERADD       EQU    $C00
10 *
11 GETIN         EQU    $FFE4
12 *
13 SIGVOL        EQU    $D418
14 ATDCY1        EQU    $D405
15 PWHI1         EQU    $D403
16 PWLO1         EQU    $D402
17 SUREL1        EQU    $D406
18 FREHI1        EQU    $D401
19 FRELO1        EQU    $D400
20 VCREG1        EQU    $D404
21 *
22 TIMER         EQU    $FA
23 CHAR          EQU    TIMER+1
24 *
25               JMP    START
26 *
27 MATRIX        DFB    62,10,9,13,18,17,21,22
28               DFB    26,29,30,34,33,37,38,42
```

Listing 15-2 cont.

```
                 DFB   45,46,50,49,53
*
HIFREQ           DFB   13,14,14,15,16,17,18,19
                 DFB   21,22,23,25,26,28,29,31
                 DFB   33,35,37,39,42
*
LOFREQ           DFB   78,24,239,210,195,195,209,239
                 DFB   31,96,181,30,156,49,223,165
                 DFB   135,134,162,223,62
*
* CLEAR SOUND REGISTERS
*
START            LDA   #0
                 STA   $FF00        ;BANK 15
*
INIT             LDA   #0
                 LDX   #$18
CLOOP            STA   $D400,X
                 DEX
                 BNE   CLOOP
*
* SET UP TIMER
*
                 LDA   #60
                 STA   TIMER
*
* SET UP INTERRUPT
*
                 LDA   CINV
                 STA   USERADD
                 LDA   CINV+1
                 STA   USERADD+1
                 SEI
                 LDA   #<WAIT
                 STA   CINV
                 LDA   #>WAIT
                 STA   CINV+1
                 CLI
*
* SET REGISTERS
*
                 LDA   #15
                 STA   SIGVOL
                 LDA   #9
                 STA   ATDCY1
                 LDA   #0
                 STA   SUREL1
                 STA   PWHI1
                 LDA   #255
                 STA   PWLO1
```

Listing 15-2 cont.

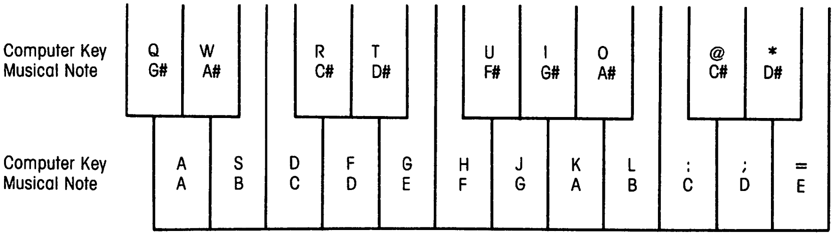
```
         LDA    #64
         STA    VCREG1
*
GETKEY   LDA    SFDX
         CMP    #88         ;WAS #64 ON C-64
         BNE    SKIP
         LDA    #0
         STA    CHAR
         JMP    GETKEY
*
SKIP     LDX    #20
CHECK    CMP    MATRIX,X
         BEQ    PLAY
         DEX
         BPL    CHECK
         JMP    GETKEY
*
PLAY     CMP    CHAR
         BNE    CONT
         JMP    GETKEY
*
CONT     STA    CHAR
         LDA    #60
         STA    TIMER
         LDA    #64
         STA    VCREG1
         LDA    HIFREQ,X
         STA    FREHI1
         LDA    LOFREQ,X
         STA    FRELO1
         LDA    #65
         STA    VCREG1
         JMP    GETKEY
*
WAIT     LDX    TIMER
         DEX
         BNE    RETURN
         LDA    #64
         STA    VCREG1
         LDX    #0
RETURN   STX    TIMER
         JMP    (USERADD)
```

We'll see how the WAIT routine works later in this chapter. For now, it's sufficient to remember that the routine ends with the statement JMP (USERADD). That statement, the first indirect jump that we have encountered in this book, ends the user-written timing loop in the MUSIC program with a jump to the address originally pointed to by the CINV vector.

Running the Program

When you type, assemble, and execute the MUSIC program, it will turn your Commodore 128's keyboard into an electronic piano. You'll be able to use the keys on the A row as white piano keys, and the keys on the Q row as black keys. Figure 15-6 shows the layout of the keys in the program.

Figure 15-6 Keyboard arrangement for the MUSIC program



How the Program Works

The source code of the MUSIC program is fairly easy to follow. In lines 41 and 42, the 8502 chip is set to access bank 15, the memory bank in which the C-128's registers appear. Then, in lines 44 through 48, SID registers $D400 through $D418 are cleared with the help of a simple X loop. Next, a note timer is set and a note-timing routine is added to the C-128's hardware interrupt (CINV) vector. Then, in lines 70 through 80, the major SID registers that control voice 1 are loaded with values that emulate the sound of a piano. (These settings, plus settings that reproduce the sounds of several other instruments, are listed on page 164 of the *Commodore 64 User's Guide.)*

The heart of the MUSIC program is a loop labeled GETKEY, which is in lines 82 through 87. This loop scans the Commodore keyboard repeatedly to see whether a key is pressed. If a key isn't pressed, the loop repeats until a keypress is detected. After a key is pressed, the program jumps to line 89, where a routine labeled SKIP begins. We'll move on to the routine labeled SKIP in a moment. First, though, let's pause to take a closer look at how the GETKEY loop works.

In line 82, the beginning of the GETKEY loop, the accumulator is loaded with the value of a memory register labeled SFDX. This register, situated at memory address $D4 (it resided at $CB in the Commodore 64), is used by a C-128 operating system routine called a keyscan interrupt routine. Sixty times every second, during the C-128's vertical blank interrupt, this memory address is loaded with a special code number that the computer uses to determine whether a key is being pressed—and, if so, what key is being pressed.

The code numbers used by the Commodore keyscan interrupt routine are neither ASCII code numbers nor Commodore screen code numbers. They are special key code numbers used only in C-64 and C-128 keyboard-scanning operations. Table 15-3 is a listing of these

Table 15-3
C-64 and C-128
Key Codes

Codes used by both the C-64 and the C-128

Key Code	Key	Key Code	Key
0	Insert/Delete	33	I
1	Return	34	J
2	Cursor right	35	0 (zero)
3	F7	36	M
4	F1	37	K
5	F3	38	O (letter)
6	F5	39	N
7	Cursor down	40	+
8	3	41	P
9	W	42	L
10	A	43	-
11	4	44	.
12	Z	45	:
13	S	46	@
14	E	47	,
15	Not used	48	+
16	5	49	*
17	R	50	;
18	D	51	Clear/Home
19	6	52	Not used
20	C	53	=
21	F	54	Up arrow (exponential sign)
22	T	55	/
23	X	56	1
24	7	57	Left arrow
25	Y	58	Not used
26	G	59	2
27	8	60	Spacebar
28	B	61	Not used
29	H	62	Q
30	U	63	Run/Stop
31	V	64	No key pressed (C-64 only)
32	9		

Codes used only by the C-128 (keypad keys and gray keys)

Key Code	Key	Key Code	Key
64	Help	77	6
65	8	78	9
66	5	79	3
67	Tab	80	Not used
68	2	81	0
69	4	82	.
70	7	83	Cursor up
71	1	84	Cursor down
72	Esc	85	Cursor left
73	+	86	Cursor right
74	-	87	No scroll
75	Line feed	88	No key pressed
76	Enter		

codes. Codes used by both the C-64 and the C-128 appear in the first part of the table; codes used by the C-128 only are in the second part of the table.

Why Key Codes Are Used in the Program

The MUSIC program uses key codes because key code values, not ASCII code values, are the kinds of values that are returned each time the Commodore 64/128 scans its keyboard to check for pressed keys. Each time a key is pressed, the Commodore operating system converts the key code value of the depressed key into an ASCII code value. Then that ASCII code value is placed in a special typeahead buffer so that it can be kept in memory long enough to be displayed on the screen.

This is a good system for printing text on a screen, but it is not an ideal system for a musical keyboard program such as MUSIC. A typeahead buffer is neither necessary nor desirable in a musical keyboard program, and the Commodore 64/128's "debounce" feature (a feature that causes a letter to be printed repeatedly on the screen after it has been held down for a short period of time) creates more problems than it solves when it is used in musical keyboard programs.

In lines 27 through 29 of the MUSIC program, a data table, labeled MATRIX, lists the key codes for all 21 keys used in the program. Immediately following this table, there are similar data tables showing the high-frequency and low-frequency code numbers of each note in the MATRIX table. In each of these tables, the offset for each note is identical. With the help of indirect addressing, therefore, the three tables can be used together to locate any valid note in the MUSIC program and to determine its proper frequency setting.

Now let's return to the GETKEY loop in the MUSIC program: the loop that begins at line 82. Notice that the loop recycles as long as the SFDX register has a value of 88. Refer to the key code table in table 15-3 and you'll see that a key code value of 88 means that no key is being pressed. (Commodore 64 programmers watch out; the code number for no key being pressed is 88 on the C-64!) As long as the SFDX register holds a value that means no key is being pressed, a value of 0 is loaded repeatedly into a special memory register that has been labeled CHAR, and the GETKEY loop in the MUSIC program keeps repeating. (We'll see in a few moments how the 0 stored in the CHAR register during this operation is used.)

As soon as a key is pressed, the value of SFDX changes from 88 to some other value, and the program jumps to the routine called SKIP that starts at line 89. This routine uses an X loop to count down through the 21 key code values that are valid in the MUSIC program. If a valid key is pressed, the program jumps to a PLAY routine that starts at line 96. Otherwise, the program keeps looping until a valid note is typed.

When the PLAY routine begins, the first thing the program does is check the status of a variable labeled CHAR. This variable determines whether a key has just been pressed or whether it is being held down. If a key has just been pressed, the CHAR register holds a different value from the value of the key being pressed. If this is the case, the program jumps to a routine labeled CONT (for "continue") and a new note is played. But if a key that has already initiated a note is still being pressed, the CHAR register holds the same value as the value of the key being pressed, and a new note does not begin.

Because of a tricky feature of the GETKEY routine, the process just described will always work, even when the same key is pressed over and over. In lines 85 and 86 of the GETKEY routine, the value of CHAR is reset to 0 every time you lift your finger from a key. Because of this feature, a key that is pressed and held down will cause a note to sound only once. But if the key is released and then pressed down again, the note will play again.

During the CONT routine, which extends from line 100 through line 111, the notes are actually played. An interrupt controlled note timer (labeled TIMER) is set to a value of 60, which corresponds to a playing time of one second. The value 64 is stored in the voice 1 control register, clearing that register's gate bit and turning off any note that may be playing. Then, the high-frequency and low-frequency codes that correspond to whatever note is selected are stored in the appropriate SID registers. Next, the value 65 is stored in the voice 1 control register, setting that register's gate bit and starting a new note. Then, an unconditional jump is made back to the GETKEY routine, so that a new note can be played. Meanwhile, the routine labeled WAIT—now a part of your computer's hardware interrupt vector—keeps ticking away, making sure that the ADSR envelope of every note you play is correctly timed.

After you type, assemble, and execute the MUSIC program, you may decide that you'd like to make it more complicated. As written, the program uses only one of the SID chip's voices—but it could easily be expanded into a program that uses all three. Then, to accompany your melody line, you could add harmony, a bass line, or even the sound of drums.

Because you've also learned how to write graphics programs in assembly language, you could improve the MUSIC program by adding some color graphics, creating something interesting to look at while the music plays. If you want to tackle a really challenging programming job, you may be able to expand the MUSIC program into one that can store and play back melodies that you've typed in on the keyboard. Then, by mixing assembly language and BASIC, you may even be able to figure out how to store selections that you've played and recorded on a disk, so that you can reload them any time you like and incorporate them into other programs.

16

Programming Sprites in Assembly Language

And more C-128 high-resolution graphics

When the Commodore 64 was introduced, one of its most exciting features was its ability to generate sprites—user-programmable graphics characters that are easy to create and animate, and can be moved independently.

Well, the Commodore 128 has sprites, too—and they are even better than the sprites built into the Commodore 64. The C-128 has a built-in sprite editor that can take much of the drudgery out of creating sprites. BASIC 7.0, the version of Commodore BASIC that's built into the C-128, has a host of new instructions for creating and animating sprites. And—best of all from an assembly language programmer's point of view—the sprites produced by the C-128 are interrupt driven; they are generated while the screen is blacked out for refreshing, and they are therefore smooth moving and flicker free when they are displayed on the screen.

In this chapter, we won't be covering the C-128's sprite editor or the sprite-related instructions that are available in BASIC 7.0; those topics are well covered in many other books, including the C-128 system guide that comes with your computer. Instead, we'll be learning how to program the C-128's sprites in assembly language—which, as every experienced assembly language programmer knows, is how sprites should be programmed.

This chapter also contains some advanced tips that can add some pizazz to high-resolution assembly language programs. You'll not only learn how to program sprites; you'll also learn how to use your C-128's built-in character set to display screen characters that are four times their normal size!

Before we can start learning and using all of these tricks, though, we'll have to explore the graphics capabilities of the Commodore 128 in a little more detail.

C-128's Video Banks

As noted in previous chapters, the Commodore 128 produces its 40-column text and graphics displays with the help of a special video chip called the 8564 Video Interface Chip II, or VIC-II. The VIC-II also controls the C-128's sprites; so sprites are available only when the C-128 is in 40-column text mode or standard high-resolution graphics mode. Sprites are not available when the C-128's 8563 VDC chip is enabled, so they cannot be used with an 80-column or double high-resolution display.

As we saw in chapter 10, the Commodore 128 has two 64K blocks of RAM—sometimes labeled RAM block A and RAM block B—and one 48K block of ROM. But, as you may also recall from previous chapters, the C-128's VIC-II chip can access only 16K of memory at a time. So, to make the VIC-II's job a little easier, the engineers who designed the C-128 divided each of its two 64K RAM blocks into four video banks, each containing 16K of memory. To make the C-128 programmer's job a little easier, they provided a simple method for telling the VIC-II which video bank to access to get the data it needs to generate a screen display.

To direct the VIC-II to the correct chip, all you have to do is set the two lowest bits in memory register $DD00, often referred to as CI2PRA. If you've had experience programming the Commodore 64, you may be familiar with the CI2PRA register. In both the C-64 and the C-128, this register selects the 16K segment of memory accessed by the VIC-II chip. And in both computers, the register's two lowest bits are read and written to using a convention known as "active low," which means that their values are inverted; they must be set to address bank 0 and cleared to address bank 3.

Figure 16-1 shows how the C-128's two blocks of RAM can be divided into four 16K video banks each. Table 16-1 shows how bits 0 and 1 of CI2PRA can direct the VIC-II chip to any desired video bank within either of the C-128's 64K blocks of RAM.

Figure 16-1
Four video banks in each memory block

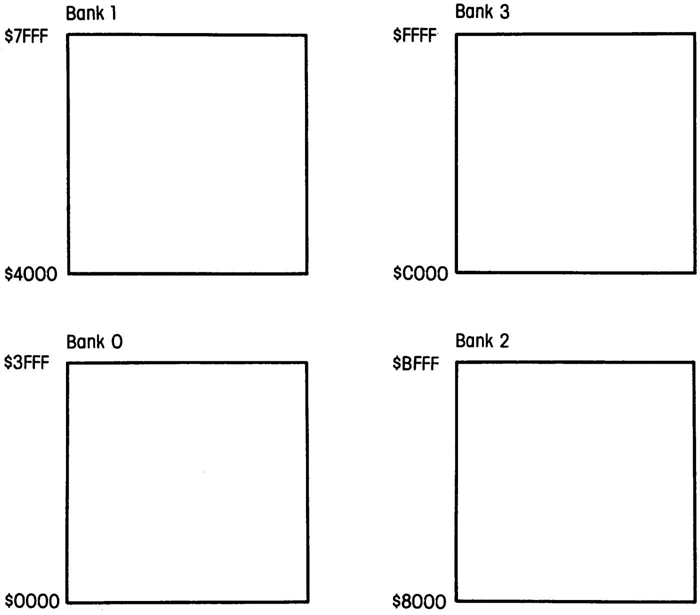


Table 16-1
Selecting a Video Bank Using $DD00 Register

Video Bank	Address Range	$DD00 Setting	Hexadecimal Equivalent
0	$0000 through $3FFF	XXXXXX11	$03
1	$4000 through $7FFF	XXXXXX10	$02
2	$8000 through $BFFF	XXXXXX01	$01
3	$C000 through $FFFF	XXXXXX00	$00

Using the CI2PRA Register

The CI2PRA is a very important register in C-128 graphics programs because it is often necessary to move the block of memory that is accessed by the VIC-II. For example, in the program at the end of this chapter—an assembly language program titled SPRITE—there are three large blocks of graphics-related data. There's a high-resolution screen, a character set that has been copied from ROM into RAM, and a sprite. Because data from each of these memory blocks appears on the screen at the same time, the C-128's VIC-II chip has to have access to all three of them simultaneously. This means that all three blocks of data have to appear in the same 16K video bank in the same 64K block of memory.

This task would not be difficult if the VIC-II chip was set to access a free 16K block of RAM when the computer is turned on. Unfortunately, this is not the case. When the C-128 is turned on, the VIC-II chip is set to access video bank 0 in RAM block A—and, as pointed out in chapter 10, that is a very crowded block of RAM. It contains page zero, the 8502 stack, and a big section of BASIC and operating system RAM—in all, more than 7K of RAM that is difficult, if not impossible, to use for storing graphics data.

Fortunately, it is not difficult to move the VIC-II's access area out of this crowded memory block and into a segment containing more free RAM. In the SPRITE program, for example, the CI2PRA chip directs the VIC-II chip to video bank 1 (memory addresses $4000 through $7000) in RAM block A. In lines 412 through 418, the 8502 chip is instructed to access memory bank 15, where the CI2PRA ($DD00) register resides. Then bits 0 and 1 of the CI2PRA register are set to access video bank 1. A masking operation is used for this procedure, as illustrated in listing 16-1.

Listing 16-1 Altering $DD00 register using masking

```
LDA CI2PRA
AND #$FC ;CLEAR BITS 0 AND 1
ORA #$02 ;USE VIDEO BANK 1
STA CI2PRA
```

Setting the Shadow VMCSB Register

Before the VIC-II chip can produce a screen display, it must also be told exactly where to go in memory to get the screen data and character data it needs to produce the screen display. In a C-128 program, screen and character data may be placed anywhere—within certain limitations. These limitations are:

- A high-resolution screen map must start on a 1K boundary—that is, at a memory address divisible by $0400, or 1024 in decimal notation.
- When a full or partial character set is copied from ROM into RAM, its starting address in RAM must be situated on a 2K boundary—that is, at a memory address divisible by $0800, or 2048 in decimal notation.

- If a RAM-based character set is used in a program, both the screen map and the relocated character set must reside in the same 16K video bank in the same 64K block of RAM.

If you have written programs for the Commodore 64, you may know that the C-64 has one memory register—often called the VMCSB register—that serves a double function in high-resolution programs. VMCSB, situated at memory address $D018, is an 8-bit register that is designed to be used as two 4-bit registers. The high nibble tells the VIC-II chip where it can find data that it needs to generate a screen map. The low nibble directs the VIC-II chip to the segment of memory that contains character data.

In the Commodore 128, the VMCSB register cannot be accessed directly from user-written programs. Instead, there are two "shadow registers," addressable from a user-written program, that can be used to pass instructions to the VMCSB register. In a program that uses a 40-column text screen, memory register $A2C is a shadow register that is used to address the VMCSB register. In a high-resolution graphics program, the VMCSB's shadow register is at memory address $A2D.

Using Memory Register $A2C

In text and low-resolution programs written for the Commodore 128, the block of memory that is usually used as a screen map extends from $0400 to $07FF in memory bank 0, and the ROM segment ordinarily used for storing character data extends from $D000 to $DFFF in bank 14. In addition, there is a color map that always occupies the segment of bank 15 memory that extends from $D800 to $DBFF.

Memory register $A2C can be used to relocate screen data, character data, or both. The high nibble of $A2C tells the VIC-II chip where it can find the screen code data it needs to produce a 40-column text screen. The low nibble of $A2C points the VIC-II to the segment of memory in which character data is stored.

For memory register $A2C to work properly, the CI2PRA ($DD00) register must be set to access the video bank in which screen and character data are stored. Tables 16-2 and 16-3 show how the VIC-II, CI2PRA, and $A2C registers can be used together to generate a text or low-resolution screen display.

Using Memory Register $A2D

When the Commodore 128 is in high-resolution mode, the block of RAM used as screen memory starts at memory address $1C00 in memory bank 0. The first 1,024 bytes of this memory block—the portion that extends from $1C00 to $1FFF—are used as a color map. The data used to bit map the screen extends from $2000 to $3FFF.

Table 16-2 Text and Low-Resolution Screen Map Addresses

Store starting address code in $A2C as follows:

		Starting Addresses			
Bits to Set	**Hex Number**	**Video Bank 0**	**Video Bank 1**	**Video Bank 2**	**Video Bank 3**
1111XXXX	$F0	$3C00	$7C00	$BC00	$FC00
1110XXXX	$E0	$3800	$7800	$B800	$F800
1101XXXX	$D0	$3400	$7400	$B400	$F400
1100XXXX	$C0	$3000	$7000	$B000	$F000
1011XXXX	$B0	$2C00	$6C00	$AC00	$EC00
1010XXXX	$A0	$2800	$6800	$A800	$E800
1001XXXX	$90	$2400	$6400	$A400	$E400
1000XXXX	$80	$2000	$6000	$A000	$E000
0111XXXX	$70	$1C00	$5C00	$9C00	$DC00
0110XXXX	$60	$1800	$5800	$9800	$D800
0101XXXX	$50	$1400	$5400	$9400	$D400
0100XXXX	$40	$1000	$5000	$9000	$D000
0011XXXX	$30	$0C00	$4C00	$8C00	$CC00
0010XXXX	$20	$0800	$4800	$8800	$C800
0001XXXX	$10	$0400	$4400	$8400	$C400
0000XXXX	$00	$0000	$4000	$8000	$C000

When the Commodore 128 is in high-resolution mode, memory register $A2D can be used to relocate both the RAM block used as a color map and the RAM block used as a bit map. The high nibble of $A2D tells the VIC-II chip where it can find the color data it needs to generate its bit-mapped screen. The low nibble directs the VIC-II to the starting address of the data that will be used to bit map the C-128's screen. Because it takes 8,000 bytes of memory to produce a bit-mapped display, however, it takes only one bit to direct the VIC-II chip to the starting address of a high-resolution screen map. Consequently, the low nibble of register $A2D has only one significant bit: bit 3.

Memory register $A2D, like memory address $A2C, works hand in hand with the CI2PRA ($DD00) register. For $A2D to work properly, the CI2PRA register must be set to access the video bank in which color data and bit-mapped data are stored. Tables 16-4 and 16-5 show how the VIC-II, CI2PRA, and $A2D registers can be used together to generate a bit-mapped high-resolution display.

Table 16-3 RAM Character Set Starting Addresses in Text Mode

Store starting address code in $A2C as follows:

		Starting Addresses			
Bits to Set	**Hex Number**	**Video Bank 0**	**Video Bank 1**	**Video Bank 2**	**Video Bank 3**
XXXX111X	$0E	$3800	$7800	$B800	$F800
XXXX110X	$0C	$3000	$7000	$B000	$F000
XXXX101X	$0A	$2800	$6800	$A800	$E800
XXXX100X	$08	$2000	$6000	$A000	$E000
XXXX011X	$06	$1800	$5800	$9800	$D800
XXXX010X	$04	$1000	$5000	$9000	$D000
XXXX001X	$02	$0800	$4800	$8800	$C800
XXXX000X	$00	$0000	$4000	$8000	$C000

Table 16-4 High-Resolution Color Map Addresses

Store starting address code in $A2C as follows:

Bits to Set	Hex Number	Starting Addresses Video Bank 0	Video Bank 1	Video Bank 2	Video Bank 3
1111XXXX	$F0	$3C00	$7C00	$BC00	$FC00
1110XXXX	$E0	$3800	$7800	$B800	$F800
1101XXXX	$D0	$3400	$7400	$B400	$F400
1100XXXX	$C0	$3000	$7000	$B000	$F000
1011XXXX	$B0	$2C00	$6C00	$AC00	$EC00
1010XXXX	$A0	$2800	$6800	$A800	$E800
1001XXXX	$90	$2400	$6400	$A400	$E400
1000XXXX	$80	$2000	$6000	$A000	$E000
0111XXXX	$70	$1C00	$5C00	$9C00	$D000
0110XXXX	$60	$1800	$5800	$9800	$D800
0101XXXX	$50	$1400	$5400	$9400	$D400
0100XXXX	$40	$1000	$5000	$9000	$D000
0011XXXX	$30	$0C00	$4C00	$8C00	$CC00
0010XXXX	$20	$0800	$4800	$8800	$C800
0001XXXX	$10	$0400	$4400	$8400	$C400
0000XXXX	$00	$0000	$4000	$8000	$C000

Table 16-5 High-Resolution Screen Map Addresses

Set bit 3 of $A2D as follows:

Setting of Bit 3	Hex Number	Starting Addresses Video Bank 0	Video Bank 1	Video Bank 2	Video Bank 3
XXXX1XXX	$08	$2000	$6000	$A000	$E000
XXXX0XXX	$00	$0000	$4000	$8000	$C000

In the SPRITE program, the block of memory used as a color map starts at $5C00, and the block used as a screen map starts at $6000. In this program, it takes only two lines of code—lines 423 and 424—to point the VIC-II chip to the two banks of memory that will be used to color map and bit map the program's high-resolution screen. Register $A2D is labeled SVMCSB (for "shadow VMCSB") in the SPRITE program; the two lines that point the VIC-II chip to the program's color map and bit map are reproduced in listing 16-2.

Listing 16-2 Setting the SVMCSB register

```
LDA #$78
STA SVMCSB
```

Creating Giant Characters

Although the SPRITE program (at the end of this chapter) was written to demonstrate the creation and animation of sprites, its most unusual feature is its ability to display giant characters on a high-resolution screen. Best of all, it accomplishes this feat without requiring the programmer to create or purchase a special character set; it simply copies the C-128's character set into RAM, and then enlarges each character to four times its normal size. Also, because each character is stored in RAM in its original size, the giant characters produced by the SPRITE program do not require a giant-sized section of memory.

Another noteworthy feature of SPRITE's character-generating module is its simplicity. To copy the C-128's character set into RAM, the program uses an algorithm much like the one used for a similar purpose in chapter 10. But, as each character is called up to be displayed on the screen, each dot is copied into screen memory twice, doubling the character's width. Each scan line in each character is also displayed twice, doubling the character's height. Result: quadruple-sized screen characters, all produced in lines 265 through 315 of the SPRITE program.

Creating a Sprite

Now that we've seen how to set up color maps and bit maps, and how to create giant-sized screen characters, we're ready to move to the topic that this chapter is supposed to be all about: creating and animating sprites in assembly language.

If you've ever worked with sprites in BASIC, you'll probably be pleased to learn that it's easier to program sprites using assembly language than it is to create them using BASIC. That's because sprites are programmed using many kinds of bit and byte manipulations that are much easier to manage using binary and hexadecimal numbers than they are using decimal numbers. By the time you finish this chapter, you'll see why.

What Is a Sprite?

Sprites, as noted at the beginning of this chapter, are graphics characters that can be created, colored, and animated quite easily, and moved independently. Using ordinary programming techniques, up to eight sprites can be displayed on a screen simultaneously. These eight sprites are usually numbered 0 through 7.

Sprites, like programmable text characters, are made of tiny dots. And, like programmable characters, they can be created using standard bit-mapping techniques. But sprites are larger than text characters; a sprite can be up to 24 horizontal screen dots wide and up to 21 vertical screen dots high.

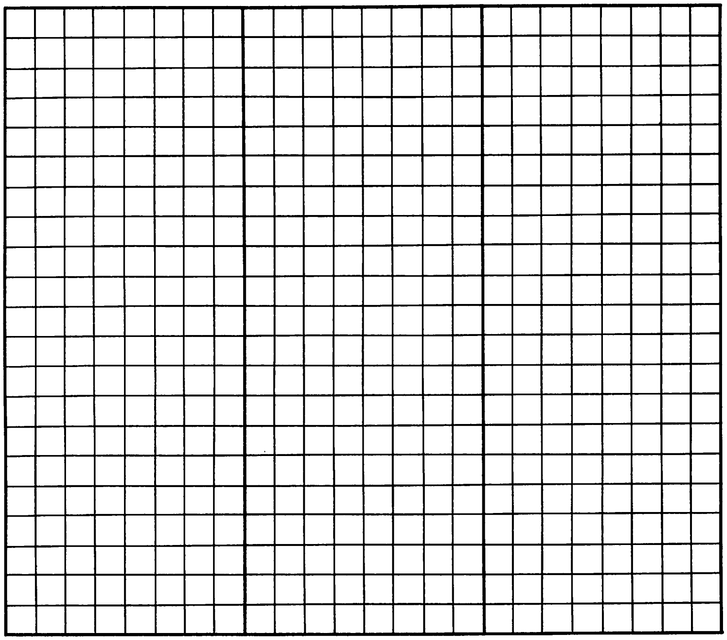
A sprite can be displayed in any of the 16 colors that are available to the VIC-II chip. It is also possible to create multicolored sprites. We won't be covering multicolored sprites in this chapter, but you can learn all about them in other books, including the system guide that comes with your computer. From an assembly language point of view, there's no difference between a multicolored sprite and any other sprite; after you've handled one sprite in assembly language, you've handled them all.

Another feature of sprites is that they can be expanded to twice their normal width and twice their normal height, or four times their standard size. The sprite used in this chapter is an expanded one.

Bit Mapping a Sprite

A sprite can measure up to 24 screen dots (or bits) wide, and up to 24 screen dots (or bits) high, for a total of 504 dots. A sprite bit map is illustrated in figure 16-2.

Figure 16-2
Sprite bit map

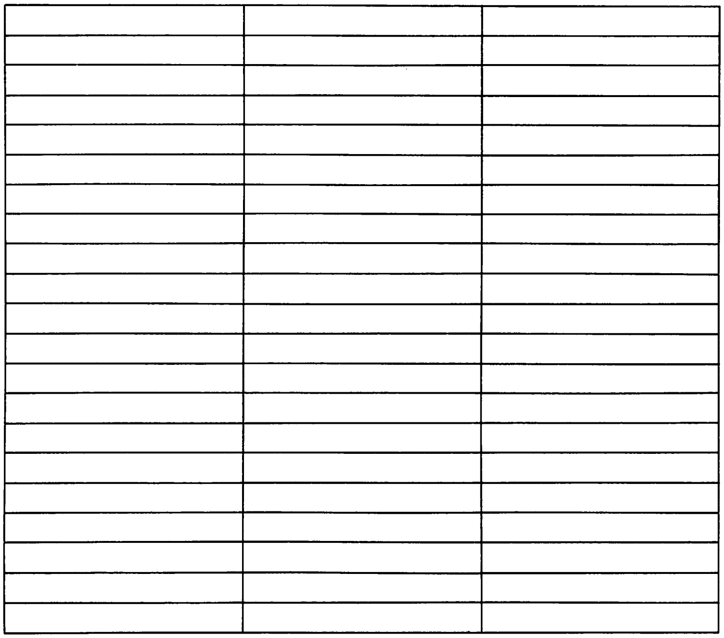


A sprite can also be pictured as a byte map—a matrix that measures 3 bytes wide by 21 bytes high, for a total of 63 bytes. Actually, the bytes that make up a sprite are in consecutive order in RAM, starting with the byte in the upper left corner and ending with the 63d byte, the one in the lower right corner. But when a sprite appears on the screen, it looks more like the byte map shown in figure 16-3.

Although it takes only 63 bytes to form a sprite, each sprite uses 64 bytes in RAM. The 64th byte of each sprite map is used to mark the end of its location in memory.

Sprites can be placed anywhere in free RAM, and a special pointer is provided to mark the location for each sprite. Each sprite pointer is one byte long, so it takes eight bytes of RAM to hold the eight pointers that are needed to address the Commodore 128's eight sprites. These eight pointers are always the last eight bytes of

Figure 16-3
Sprite byte map



whatever block of RAM has been designated as screen memory. When the location of screen memory is moved, the addresses of the C-128's eight sprite pointers also change. But it's always easy to find these addresses, because they're always the last eight bytes of whatever block of RAM is being used as screen memory.

You only need a 1-byte value to define the starting address of a sprite map, because sprites are always in whatever 16K bank of memory is currently accessible to the VIC-II chip. This means a sprite pointer is actually an offset that must be added to the starting address of the video bank currently in use to determine the starting address of the bit map that forms the sprite.

As we have seen, when the Commodore 128 is first turned on, its VIC-II chip is set to retrieve graphics information from video bank 0 in RAM block A, and to get its screen map from memory registers $0400 through $07FF (1024 through 2047 in decimal notation). Therefore, when the computer is first turned on, the default address of the first sprite pointer, or sprite pointer 0, is $07F8 (or 2040 in decimal notation). The next eight bytes in RAM are the pointers for sprites 1 through 7. So the default addresses of the pointers for the C-128's eight sprite pointers are memory addresses $07F8 through $07FF—the last eight bytes in the block of RAM designated as screen memory.

Therefore, to find the data it needs to display a sprite, the Commodore 128 only has to look at the 8-bit value stored in the appropriate sprite pointer. When that value is added to the address of the graphics bank currently in use, the result is the address of the bit map that defines the sprite.

Turning Sprites On and Off

Before a sprite can be displayed, it must be turned on. Sprites are turned on and off with a sprite enable register (abbreviated SPENA) at memory address $D015 in memory bank 15. Each bit of the SPENA register is associated with one sprite; bit 0 turns sprite 0 on and off, bit 1 controls sprite 1, bit 2 controls sprite 2, and so on. If the bit associated with a sprite is set, then the sprite is enabled. If the bit is not set, then the sprite is disabled and cannot be used.

Positioning Sprites

Each of the C-128's eight sprites has two position registers: an X position register that determines the sprite's horizontal placement on the screen, and a Y position register that determines the sprite's vertical position. These registers are abbreviated SP0X through SP7X and SP0Y through SP7Y, respectively. In addition, there is a special most significant X position register (abbreviated MSIGX) that designates the horizontal positions of all eight sprites. We need this register because a sprite can be placed in 512 possible horizontal screen positions—too many positions for an 8-bit register to keep track of. If a sprite is placed in a position that can be stored as a value in an 8-bit register—that is, in a position with a value less than 255—then the MSIGX register is not used. But if the horizontal position of a sprite has a value of more than 255, a bit in the MSIGX register is set. Each bit of the MSIGX register equates to the number of a sprite; bit 0 is used for sprite 0, bit 1 is used for sprite 1, and so on.

There is no MSIGY register because we don't need one. A sprite can be placed in only 256 vertical positions, so only one 8-bit register per sprite is needed to handle the vertical positioning of sprites on the C-128's screen.

"Shadow" Position Registers

In the Commodore 128, the memory registers that determine the screen positions of sprites are situated at addresses $D000 through $D010 in memory bank 15—the same addresses they occupy in the C-64's memory. When the proper values are stored in a horizontal or vertical position sprite register, the C-128—like the C-64—uses those values to determine the position of the upper left corner of the sprite.

There is an important difference, though, in the way the sprite position registers are used in the Commodore 64 and the way they are used in the Commodore 128. In the C-128, the values that determine sprite positions cannot be placed directly into memory registers $D000 through $D010. Every 1/60 of a second, during the C-128's vertical blank interrupt, the contents of the sprite position registers are erased and replaced by the contents of a block of shadow registers at memory addresses $11D6 through $11E6. So, when you use sprites in a C-128 program, you must set their positions on the screen using memory registers $11D6 through $11E6 rather than using memory registers $D000 through $D00F.

Moving Sprites off the Screen

Another important fact about sprite positions is that storing a value in a horizontal or vertical position register does not ensure that a sprite will be displayed on the screen. Of the 512 possible horizontal positions for a sprite, only positions 24 through 343 are visible on the screen. Of the 255 available vertical positions, only positions 50 through 249 are visible on the screen. It's therefore quite easy to make a sprite disappear; just store the value of an offscreen position in the sprite's horizontal or vertical position register.

Table 16-6 shows the shadow position registers that must be used to position each of the C-128's sprites horizontally and vertically on the screen.

Table 16-6
Sprite Position Registers

Hex Address	Position Register	Hex Address	Position Register
$11D6	SP0X	$11DF	SP4Y
$11D7	SP0Y	$11E0	SP5X
$11D8	SP1X	$11E1	SP5Y
$11D9	SP1Y	$11E2	SP6X
$11DA	SP2X	$11E3	SP6Y
$11DB	SP2Y	$11E4	SP7X
$11DC	SP3X	$11E5	SP7Y
$11DD	SP3Y	$11E6	MSIGX*
$11DE	SP4X		

*Most significant X position register

Selecting Colors for Sprites

In addition to its two position registers and its one bit in the most significant bit registers, each sprite also has a color register. The color register for sprite 0 is at memory address $D027 in memory bank 15, and the addresses of the color registers for the other seven sprites follow in consecutive order. The color address for sprite 7 is therefore at memory address $D02E in memory bank 15.

To select the color of a sprite, just store the standard value of one of the Commodore 128's 16 colors in that sprite's color register. Every bit that is set to 1 on the sprite's bit map is displayed in the selected color. Every dot that has a value of 0 is transparent, and does not cover up anything that is beneath it on the screen.

Expanding Sprites

As previously mentioned, a sprite normally measures 24 horizontal screen dots wide by 21 vertical screen dots high. But by using two special registers called XXPAND and YXPAND, a sprite can be expanded to twice its normal width, twice its normal height, or both. The XXPAND register is at memory address $D01D in memory bank 15 and the YXPAND register is at $D017 in memory bank 15. Each bit in each register corresponds to a sprite number, with bit 0 controlling the size of sprite 0, bit 1 controlling the size of sprite 1, and so on.

On with the SPRITE Program

Now we're ready to take a look at SPRITE: an assembly language program that uses high-resolution graphics, an alternate character set, a large-type printing routine, and an animated, expanded sprite routine. The program copies a character set from ROM into RAM and then prints a message on the screen in large type. It then clears a bit map for sprite 0, copies some data into the bit map from the character set in RAM, and places an expanded sprite in an area out of viewing range at the top of the screen. Next, the sprite descends into viewing range, and maintains a slow descent until it reaches a predetermined position. Then it stops, and becomes part of the message displayed on the screen.

One noteworthy feature of the SPRITE program is its use of a kernel routine called INDFET, which has a call address of $FF74. With the help of the INDFET routine, a program can load the accumulator with any value from any of the C-128's 16 memory banks, without leaving the memory bank that is currently active.

The INDFET routine works similar to indirect indexed addressing. Indirect indexed addressing, as you may recall from chapter 6, is a form of addressing in which the Y register and a 2-byte zero page pointer are used in the following format:

LDA (*pointer*),Y

where *pointer* is a zero page pointer.

Before indirect indexed addressing is used in a program, a base address must be placed in a 2-byte zero page pointer and an index value must be placed in the Y register. Then, when a statement that uses indirect indexed addressing is encountered, the value stored in the Y register is added to the 8-bit address pointed to by the pointer, and the accumulator is loaded with the contents of the resulting address.

To use INDFET, you have to do the following. Store a base address in a zero page pointer, load the accumulator with the address of the pointer, load the X register with the desired bank number, and load the Y register with an index. Then you can load the accumulator with any value you want by simply doing a JSR to memory address $FF74.

INDFET is not the only routine in the C-128's kernel that was designed to handle communications between one memory bank and another; there are other kernel routines that can place values in other banks, compare values in different banks, and perform jumps from one bank to another. Descriptions of these routines, and all of the other routines in the C-128 kernel, can be found in appendix B.

Listing 16-3
SPRITE program

```
*
* SPRITE
*
          ORG    $1300
*
COLOR     EQU    $E0
*
TABLEN    EQU    $800
MAPLEN    EQU    1000
SCRLEN    EQU    8000
SPOADR    EQU    $4E00
COLMAP    EQU    $5C00
NEWADR    EQU    $4000  ;COLOR MAP
*                       ;NEW CHARACTER SET
SPRPTR    EQU    $5FF8
VIDEO MATRIX            ;LAST 8 BYTES OF
SPENA     EQU    $D015
SPOCOL    EQU    $D027
SPOX      EQU    $11D6
SPOY      EQU    $11D7  ;SHADOW ADDRESS
MSIGX     EQU    $11E6  ;DITTO
YXPAND    EQU    $D017  ;THIS ONE, TOO
XXPAND    EQU    $D01D
*
INDFET    EQU    $FF74
*
HMAX      EQU    320
VMID      EQU    100-8
*
R6510     EQU    $0001
BASE      EQU    $6000
CHRBAS    EQU    $D000   ;NEW SCREEN MAP
SCROLY    EQU    $D011
SVMCSB    EQU    $A2D
BORDER    EQU    $D020
CIACRE    EQU    $DC0E
CI2PRA    EQU    $DD00
CIADIR    EQU    $DD02
*
TEMPA     EQU    $C8
TEMPB     EQU    TEMPA+2
TABPTR    EQU    TEMPA
*
MVSRCE    EQU    $FA
```

Listing 16-3 cont.

```
MVDEST   EQU   MVSRCE+2
BYTPTR   EQU   MVDEST+2
*
TABSIZ   EQU   $0C00
*
HPSN     EQU   TABSIZ+2
VPSN     EQU   HPSN+2
CHAR     EQU   VPSN+1
ROW      EQU   CHAR+1
LINE     EQU   ROW+1
BYTE     EQU   LINE+1
BITT     EQU   BYTE+2
*
MPRL     EQU   BITT+1
MPRH     EQU   MPRL+1
MPDL     EQU   MPRH+1
MPDH     EQU   MPDL+1
PRODL    EQU   MPDH+1
PRODH    EQU   PRODL+1
FILVAL   EQU   PRODH+1
LENPTR   EQU   FILVAL+1
CHCODE   EQU   LENPTR+2
HPTR     EQU   CHCODE+2
VPTR     EQU   HPTR+2
ONEBYT   EQU   VPTR+1
COUNT    EQU   ONEBYT+2
LTTR     EQU   COUNT+1
*
         JMP   START
*
TEXT     DFB   9,32,32,13,25,32,3,15
         DFB   13,13,15,4,15,18,5,32
         DFB   49,50,56,0
*
* BLOCK FILL ROUTINE
*
BLKFIL   LDA   FILVAL
         LDX   TABSIZ+1
         BEQ   PARTPG
         LDY   #0
FULLPG   STA   (TABPTR),Y
         INY
         BNE   FULLPG
         INC   TABPTR+1
         DEX
         BNE   FULLPG
PARTPG   LDX   TABSIZ
         BEQ   FINI
         LDY   #0
```

Listing 16-3 cont.

```
PARTLP   STA   (TABPTR),Y
         INY
         DEX
         BNE   PARTLP
FINI     RTS
*
* 16-BIT MULTIPLICATION ROUTINE
*
MULT16   LDA   #0
         STA   PRODL
         STA   PRODH
         LDX   #17
         CLC
MULT     ROR   PRODH
         ROR   PRODL
         ROR   MPRH
         ROR   MPRL
         BCC   CTDOWN
         CLC
         LDA   MPDL
         ADC   PRODL
         STA   PRODL
         LDA   MPDH
         ADC   PRODH
         STA   PRODH
CTDOWN   DEX
         BNE   MULT
         RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8 (8-BIT DIVIDE)
*
PLOT     LDA   VPSN
         LSR   A
         LSR   A
         LSR   A
         STA   ROW
*
* CHAR=HPSN/8 (16-BIT DIVIDE)
*
         LDA   HPSN
         STA   TEMPA
         LDA   HPSN+1
         STA   TEMPA+1
         LDX   #3
DLOOP    LSR   TEMPA+1
         ROR   TEMPA
         DEX
```

Listing 16-3 cont.

```
          BNE   DLOOP
          LDA   TEMPA
          STA   CHAR
*
* LINE=VPSN AND 7
*
          LDA   VPSN
          AND   #7
          STA   LINE
*
* BIT=7-(HPSN AND 7)
*
          LDA   HPSN
          AND   #7
          STA   BITT
          SEC
          LDA   #7
          SBC   BITT
          STA   BITT
*
* BYTE=BASE+ROW*HMAX+8*CHAR+LINE
*
* FIRST MULTIPLY ROW * HMAX
*
          LDA   ROW
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   #<HMAX
          STA   MPDL
          LDA   #>HMAX
          STA   MPDH
          JSR   MULT16
          LDA   MPRL
          STA   TEMPA
          LDA   MPRL+1
          STA   TEMPA+1
*
* ADD PRODUCT TO BASE
*
          CLC
          LDA   #<BASE
          ADC   TEMPA
          STA   TEMPA
          LDA   #>BASE
          ADC   TEMPA+1
          STA   TEMPA+1
*
* MULTIPLY 8 * CHAR
```

Listing 16-3 cont.

```
*
         LDA   #8
         STA   MPRL
         LDA   #0
         STA   MPRH
         LDA   CHAR
         STA   MPDL
         LDA   #0
         STA   MPDH
         JSR   MULT16
         LDA   MPRL
         STA   TEMPB
         LDA   MPRH
         STA   TEMPB+1
*
* ADD LINE
*
         CLC
         LDA   TEMPB
         ADC   LINE
         STA   TEMPB
         LDA   TEMPB+1
         ADC   #0
         STA   TEMPB+1
*
* TEMPA + TEMPB = BYTE
*
         CLC
         LDA   TEMPA
         ADC   TEMPB
         STA   TEMPB
         LDA   TEMPA+1
         ADC   TEMPB+1
         STA   TEMPB+1
*
*POKE BYTE,PEEK(BYTE)OR2↑BIT
*
         LDX   BITT
         INX
         LDA   #0
         SEC
SQUARE   ROL
         DEX
         BNE   SQUARE
         LDY   #0
         ORA   (TEMPB),Y
         STA   (TEMPB),Y
         RTS
*
```

Listing 16-3 cont.

```
* CALCULATE CHCODE'S ADDRESS
*
GETADR   LDA   #0
         STA   CHCODE+1
         LDA   CHCODE
         CLC
         ASL   A
         ROL   CHCODE+1
         ASL   A
         ROL   CHCODE+1
         ASL   A
         ROL   CHCODE+1
         STA   CHCODE
*
         CLC
         LDA   CHCODE
         ADC   #<NEWADR
         STA   BYTPTR
         LDA   CHCODE+1
         ADC   #>NEWADR
         STA   BYTPTR+1
         RTS
*
* DRAW A CHARACTER
*
DRAWCH   LDA   LTTR
         STA   CHCODE
         JSR   GETADR
*
* A NESTED LOOP:
*
* (X IS THE OUTSIDE LOOP)
*
         LDX   #8
*
* SET UP COUNTER FOR 2 VERT LINES
*
SETLIN   LDA   #2
         STA   COUNT
*
DRAWLN   LDY   #0
         LDA   (BYTPTR),Y
         STA   ONEBYT
*
* THE INSIDE LOOP:
*
* (Y IS ZERO AT START)
*
RSHIFT   LDA   ONEBYT
```

Listing 16-3 cont.

```
                ASL    A
                STA    ONEBYT
                BCS    SHOW
*
                INC    HPSN
                BNE    ITSOK
                INC    HPSN+1
ITSOK           JMP    NOSHOW
*
* DISPLAY BIT
*
* SAVE X AND Y REGISTERS
*
SHOW            TXA
                PHA
                TYA
                PHA
*
                JSR    PLOT
*
* NOW DO IT AGAIN
*
                INC    HPSN
                BNE    NOINC
                INC    HPSN+1
*
NOINC           JSR    PLOT
*
* RETRIEVE X AND Y REGISTERS
*
                PLA
                TAY
                PLA
                TAX
*
NOSHOW          INC    HPSN
                BNE    LEAP
                INC    HPSN+1
*
LEAP            INY
                CPY    #8
                BCC    RSHIFT
*
                INC    VPSN
*
                LDA    HPTR
                STA    HPSN
                LDA    HPTR+1
                STA    HPSN+1
```

Listing 16-3 cont.

```
*
* 2 VERT LINES DONE YET?
*
         DEC   COUNT
         BNE   DRAWLN
*
         INC   BYTPTR
         BNE   OKMSB
         INC   BYTPTR+1
OKMSB    DEX
         BNE   SETLIN
         RTS
*
* SUBROUTINE TO COPY CH R SET INTO RAM
*
* POKE CHR DATA INTO NEW LOCATION
*
COPYCHRS LDA   #<CHRBAS
         STA   MVSRCE
         LDA   #>CHRBAS
         STA   MVSRCE+1
*
         LDA   #<NEWADR
         STA   MVDEST
         LDA   #>NEWADR
         STA   MVDEST+1
*
         LDA   #<TABLEN
         STA   LENPTR
         LDA   #>TABLEN
         STA   LENPTR+1
*
         LDY   #0
         LDX   LENPTR+1
         BEQ   MVPART
MVPAGE   JSR   GETDATA
         INY
         BNE   MVPAGE
         INC   MVSRCE+1
         INC   MVDEST+1
         DEX
         BNE   MVPAGE
MVPART   LDX   LENPTR
         BEQ   MVEXIT
MVLAST   JSR   GETDATA
         INY
         DEX
         BNE   MVLAST
MVEXIT   RTS
```

Listing 16-3 cont.

```
*
* SUBROUTINE TO STORE (MVSRCE),Y IN (MVDEST),Y
*
GETDATA  PHA          ;SAVE ACCUMULATOR
         TXA          ;SAVE X REGISTER
         PHA
         LDA   #MVSRCE
         LDX   #14     ;GET CHR DATA IN BANK 14
         JSR   INDFET
         STA   (MVDEST),Y ;STORE IT IN BANK 15
         PLA
         TAX          ;RESTORE X REGISTER
         PLA          ;RESTORE ACCUMULATOR
         RTS
*
*
* MAIN ROUTINE STARTS HERE
*
START    JSR   COPYCHRS    ;COPY CHR SET INTO RAM
*
         LDA   #$20
         STA   $D8 ;ACTIVATE HIGH-RES GRAPHICS
*
* USE VIDEO BANK 1 ($4000-$7FFF)
*
         LDA   $0
         STA   $FF00       ;CI2PRA IS IN BANK 15
         LDA   CI2PRA
         AND   #$FC        ;%11111100
         ORA   #$02
         STA   CI2PRA
         STA   $FF01  ;RETURN TO BANK 0
*
* PUT SCREEN MAP AT $6000,
* COLOR MAP AT $5C00
*
         LDA   #$78
         STA   SVMCSB
*
* CLEAR BIT MAP
*
         LDA   #0
         STA   FILVAL
```

Listing 16-3 cont.

```
                LDA   #<BASE
                STA   TABPTR
                LDA   #>BASE
                STA   TABPTR+1
                LDA   #<SCRLEN
                STA   TABSIZ
                LDA   #>SCRLEN
                STA   TABSIZ+1
                JSR   BLKFIL
*
* SET LINE, BKG AND BORDER COLORS
*
                LDA   #COLOR
                STA   FILVAL
                LDA   #<COLMAP
                STA   TABPTR
                LDA   #>COLMAP
                STA   TABPTR+1
                LDA   #<MAPLEN
                STA   TABSIZ
                LDA   #>MAPLEN
                STA   TABSIZ+1
                JSR   BLKFIL
                LDA   #13         ;GREEN
                STA   BORDER
*
* POSITION MESSAGE ON SCREEN
*
                LDA   #8          *
                STA   HPSN
                STA   HPTR
                LDA   #0          *
                STA   HPSN+1
                STA   HPTR+1
                LDA   #VMID
                STA   VPSN
                STA   VPTR
*
* PRINT LINE OF LARGE TYPE
*
                LDX   #0
DISP            LDA   TEXT,X
                CMP   #0          ;EOF
                BEQ   DONE
                STA   LTTR
                TXA
                PHA
                JSR   DRAWCH
                PLA
```

Listing 16-3 cont.

```
         TAX
*
* ADVANCE CURSOR
*
         CLC
         LDA   HPTR
         ADC   #16
         STA   HPTR
         STA   HPSN
         LDA   HPTR+1
         ADC   #0
         STA   HPTR+1
         STA   HPSN+1
         LDA   VPTR
         STA   VPSN
*
* PRINT NEXT LETTER
*
         INX
         JMP   DISP
*
DONE     NOP
*
* DISPLAY SPRITE #0
*
* DEFINE SPRITE
*
* CLEAR SPRITE MAP
*
         LDA   #$00
         STA   FILVAL
         LDA   #<SPOADR
         STA   TABPTR
         LDA   #>SPOADR
         STA   TABPTR+1
         LDA   #64
         STA   TABSIZ
         LDA   #0
         STA   TABSIZ+1
         JSR   BLKFIL
*
* COPY HEART CHR INTO SPRITE DATA BLOCK
*
         LDA   #<SPOADR
         STA   TEMPA
         LDA   #>SPOADR
         STA   TEMPA+1
         LDA   #83         ;HEART
         STA   CHCODE
```

Listing 16-3 cont.

```
         JSR   GETADR
         LDY   #0
         LDX   #8
*
DEFSPO   LDA   (BYTPTR),Y
         STA   (TEMPA),Y
*
         INC   BYTPTR
         INC   TEMPA
         INC   TEMPA
         INC   TEMPA
*
         DEX
         BNE   DEFSPO
*
* PLACE SPRITE'S ADDRESS IN SPRITE POINTER
*
         LDA   #$38
         STA   SPRPTR
*
         LDA   #0
         STA   $FF00       ;GOING TO DO SOME WORK IN BANK 15
*
* EXPAND SPRITE (VERT & HORZ)
*
         LDA   #1
         STA   XXPAND
         STA   YXPAND
*
* TURN ON SPRITE #0
*
         LDA   #1
         STA   SPENA
*
* MAKE SPRITE RED
*
         LDA   $10         ;RED
         STA   SPOCOL
*
         STA   $FF01  ;RETURN TO BANK 0
*
* POSITION SPRITE ON SCREEN
*
         LDA   #54
         STA   SPOX
         LDA   #0
         STA   MSIGX
```

Listing 16-3 cont.

```
             LDA    #34
             STA    SPOY
*
* MOVE SPRITE DOWN SCREEN
*
DROP         INC    SPOY
*
* DELAY LOOP
*
             LDX    #$FF
XLOOP        LDY    #$10
YLOOP        DEY
             BNE    YLOOP
             DEX
             BNE    XLOOP
*
             LDA    SPOY
             CMP    #142
             BNE    DROP
*
INF          JMP    INF         ;INFINITE LOOP
```

17

The 80-Column Commodore

Programming text and graphics on the C-128's 80-column screen

It is no secret that the Commodore 128 can produce a terrific looking 80-column screen, with a sharp, clear set of text characters that can be displayed in up to 16 bright colors on an 80-column-by-25-row display. But there are other facts about the C-128's 80-column screen that are not as widely known. For example:

- The C-128 produces its 80-column display with the help of a special video chip that can also generate a double high-resolution graphics display. When the C-128 is in double high-resolution mode, it can display only two colors at a time, but with twice the resolution of the C-128's standard high-resolution graphics mode: 640-by-400 dots, each of which can be individually controlled in an assembly language program.
- Even when the C-128 is in standard 80-column text mode, some little-known programming techniques can be used to create interesting and useful effects. For instance, the C-128's 80-column character set can be modified in almost any way desired, and can even be reduced or enlarged to almost any size.

You'll learn how to program the C-128's 80-column video chip in both text mode and graphics mode. You'll also learn how to write an assembly language program that allows the user to type messages on the C-128's 80-column screen in characters that are twice their normal size. First, though, let's take a little time to see how the Commodore 128 produces its 80-column display.

80-Column Video Chip: 8563 VDC

The microprocessor that is used to generate the C-128's 80-column display is a special chip called the 8563 Video Display Chip, or 8563 VDC. The 8563 VDC chip is completely different from the 8564 VIC-II, the video chip that generates the C-128's 40-column text and graphics displays. When the Commodore 128 is placed in 80-column mode, its 40-column VIC-II chip is switched out of the system completely and the 8563 chip is switched in. Both of the C-128's built-in character sets are copied from ROM into the 8563's RAM, and the 8563 is then ready to generate a screen display.

To take full advantage of the 8563's features, you must have your C-128 connected to an 80-column color or monochrome monitor. A color monitor is recommended because the 8563, when connected to an RGB monitor or a specially designed 80-column composite monitor, can generate a spectacular 16-color text display.

Good News and Bad News

The 8563 chip has 16K of built-in RAM and all kinds of special features, but it is a rather difficult chip to access from an assembly language program. That's because the 8563 chip is linked to just two memory addresses on the C-128 memory map: $D600 and $D601 in bank 15. So every time an instruction must be issued to the 8563 chip, it must be funneled through these two memory registers. Every time a byte of data must be read from the 8563 chip, it must also pass through this same bottleneck.

This is a very efficient arrangement in terms of memory management, because it makes 16K of memory space available through only two bytes of RAM. But it is such a costly arrangement in terms of speed that the 8563 is considered primarily a text display chip, not a chip for generating screen displays for high-speed arcade-style games.

Features of the 8563 Chip

The 8563 chip, as mentioned, has 16K of built-in RAM and 16K of internal RAM. This block of RAM is laid out in the 8563 as illustrated in table 17-1.

Table 17-1
8563 Chip's Internal RAM Layout

Addresses within 8563 Chip	Contents of RAM
$00000 through $07FF	Screen display area (screen map)
$0800 through $0FFF	Character attributes (colors and features of text characters)
$2000 through $3FFF	Character definitions (character generator data)

When you start writing 80-column programs for the C-128, it is important to remember that the VDC's screen map, character generator data, and special-purpose registers have no specific addresses on the C-128's memory map. Instead, as noted earlier, the VDC must be addressed through memory addresses $D600 and $D601. Let's see what we can do and then we'll see how to do it.

Briefly, you can do almost anything with an 80-column display that you can do with a 40-column display, and then some. You can display characters in reverse video, in underlined mode, or in overlined mode. You can move the screen display up or down, or to the left or right, using fine scrolling. You can place text within a "frame," or window, anywhere you like on the screen. You can even redefine the looks of the VDC's cursor! Unfortunately, though, the VDC has no capability for producing sprites. Oh, well, you can't have everything.

How the 8563 Chip Works

The 8563 performs all of its special functions through 37 special registers numbered 0 through 36. To understand all of the features of these 37 registers, you need to be very proficient at video chip programming, and you need at least a basic knowledge of such things as vertical blanking interrupts. It is beyond the scope of this book to explain principles like these in any great detail, but the names and functions of the VDC's 37 internal registers are listed in table 17-2.

Table 17-2
8563 Chip's
Internal Registers

Register	Default Setting	Name and Function
R0	$7E	Horizontal total. The number of characters, minus 1, between successive horizontal sync pulses. Setting is based in part on monitor specifications.
R1	$50	Horizontal displayed. The number of characters in each horizontal row.
R2	$66	Horizontal sync position. The number of characters from the displayed part of a horizontal row to the start of the horizontal sync pulse.
R3	$49	Horizontal and vertical sync width. Bits 0 through 3 hold the width of the horizontal sync pulse in characters, plus 1. Bits 4 through 7 hold the width of the vertical sync pulse in scan lines.
R4	$27	Vertical total. The number of character rows, minus 1, between vertical sync pulses.
R5	$E0	Vertical total adjust. The number of scan lines added to the end of the display frame for adjustment of the vertical sync rate.
R6	$19	Vertical displayed. The number of characters displayed in a frame. This register sets the height of the frame.
R7	$20	Vertical sync position. The number of character rows, plus one, from the first displayed character row to the start of the vertical sync pulse.
R8	$FC	Interlace mode. Used to define the interlace mode of screen characters; that is, how scan lines are combined to produce the characters on the screen.
R9	$E7	Character total (vertical). The number of scan lines, minus one, used to create each character on the screen.
R10	$A0	Cursor start scan line and cursor mode. Bits 0 through 4 determine the top scan line used for display of the cursor. Bits 5 through 6 set the characteristics of the cursor, as follows: 00 = nonflashing 01 = invisible cursor 10 = fast flashing 11 = normal flashing
R11	$E7	Cursor end scan line. Bits 0 through 4 determine the number of the scan line at which the bottom line of the cursor will be displayed. Bits 5 through 7 are always set.
R12	$00	Display start address (high). The high byte of the start of video RAM in the VDC's memory.
R13	$00	Display start address (low). The low byte of the start of video RAM in the VDC's memory.
R14		Cursor position (high). The high byte of the cursor's current position on the 8563's screen map.

Table 17-2 cont.

Register	Default Setting	Name and Function
R15		Cursor position (low). The low byte of the cursor's current position on the 8563's screen map.
R16		Light pen (vertical). The vertical position of a light pen on the C-128's screen.
R17		Light pen (horizontal). The horizontal position of a light pen on the C-128's screen.
R18		Update address (high). Used to communicate with the 8563 chip through addresses $D600 and $D601. The high byte of the address to be manipulated is stored in this register.
R19		Update address (low). Used to communicate with the 8563 chip through addresses $D600 and $D601. The low byte of the address to be manipulated is stored in this register.
R20	$04	Attribute address (high). The high byte of the starting address of the character attribute block of the VDC's memory.
R21	$00	Attribute address (low). The low byte of the starting address of the character attribute block of the VDC's memory.
R22	$78	Character and total displayed (horizontal). Bits 4 through 7 determine the total number of horizontal pixels in each screen character. Bits 0 through 3 determine the number of displayed horizontal pixels in each character.
R23	$E8	Character displayed (vertical). The number of vertical scan lines displayed for each character. This register determines the height of screen characters.
R24	$20	Vertical smooth scroll. Used to control vertical smooth scrolling of the screen display. In addition, this register can control the flashing rate of characters, reverse the foreground and background screen colors, and manage block-copying operations.
R25	$40	Horizontal smooth scroll. Used to control horizontal smooth scrolling of the screen display. This register is also used to select between the VDC's text mode or double high-resolution graphics mode, to enable or disable character attributes, and to determine whether a 40-character screen or an 80-column screen will be displayed.
R26	$F0	Foreground and background color. Bits 0 through 3 determine the screen background color. Bits 4 through 7 determine the color for set bits on the screen map when character attributes are disabled.
R27	$00	Address increment row. The number of bytes to be added to video RAM for each displayed column. Used to program fine scrolling.

Table 17-2 cont.

Register	Default Setting	Name and Function
R28	$2F	Character base address and RAM type. Bits 5 through 7 determine the starting address of character data in the 8563's memory. Bit 4 determines the type of RAM configurations that the 8563 can address.
R29	$E7	Underline scan line. The number of the scan line at which an underline can appear. This register can be used to change underlining to overlining.
R30		Word count. The number of characters to be written to the update address stored in R18 and R19, or to be copied from one address to another.
R31		CPU data register. Contains data to be written to a register or data that has been read from a register.
R32		Block copy start address (high). The high byte of the starting address of a block of memory to be copied.
R33		Block copy start address (low). The low byte of the starting address of a block of memory to be copied.
R34	$7D	Display enable (begin). The number of characters on a line from the first displayed character in a row to the first blanked character in the same row. This register can be used with R35 to set up frames, or windows, of any width on the screen.
R35	$40	Display enable (end). The number of characters on a line from the first displayed character in a row to the last blanked character in the same row. This register can be used with R34 to set up frames, or windows, of any width on the screen.
R36	$F5	RAM refresh rate. Bits 0 through 3 specify the rate at which the 8563's RAM is refreshed during raster interrupts.

Programming the 8563 Register

Now that we have examined the functions of the VDC's 37 internal registers, we are ready to see how the contents of these registers can be read and manipulated in a C-128 program. After you know how to write to, and read, the VDC's internal registers, you'll know all that you need to know to write 80-column programs for the Commodore 128.

Writing to an 8563 Register

To write a value to an 8563 register, you have to do the following:

1. Load the accumulator with the number of the register that you want to address.

2. Store the number of the desired register in memory address $D600.
3. When the number of the desired register is stored in memory address $D600, the VDC responds by clearing bit 7 of memory address $D600. Then, a few nanoseconds later, the VDC sets bit 7 to let you know that the value it has received has been accepted.
4. When the VDC has accepted the value passed to it and has set bit 7 of memory address $D600, you can store whatever value you like in the VDC register that you have chosen. You must store the value in memory address $D601. The VDC then ensures that the desired value is written to the desired register.

Listing 17-1, a program titled CURSOR.SRC, demonstrates how the 8563's internal registers can be accessed in an assembly language program. CURSOR.SRC, as we shall see later in this chapter, modifies the @ character in the 8563's character set into an underline cursor for use in another program. The portion of the CURSOR.SRC program that we're interested in now is the segment labeled WRITE, which starts at line 1270. When the WRITE routine begins, the address of the VDC register to be addressed is stored in the X register, and the value to be written to the chosen register is stored in the accumulator.

In line 1280, the program uses a BIT instruction and a loop labeled WAIT to see whether the 8563 has accepted the value passed to it through memory address $D600. The loop ends when the VDC accepts the value passed to it and sets bit 7 of memory register $D700. When that happens, the BPL instruction in line 1290 detects the set bit and the WAIT loop ends. Then the value to be written to the chosen register is stored in memory address $D601, and the routine ends.

Listing 17-1 CURSOR.SRC program

```
1000 ;
1010 ; CURSOR.SRC
1020 ;
1030  *=$0D00
1040 ;
1050 VDC = $D600
1060 ;
1070  JMP START
1080 ;
1090 SHAPE .BYT $00,$00,$00,$00,$00,$00,
     $00,$FF
1100 ;
1110 START LDX #18 ;ADDRESS UPDATE
     REGISTER, HIGH BYTE
1120  LDA #$20 ;HIGH BYTE OF ADDRESS TO BE
      CHANGED IS $20
1130  JSR WRITE
```

Listing 17-1 cont.

```
1140  INX ;NOW THE VALUE OF X IS #19
      (ADDRESS UPDATE REG, LOW BYTE)
1150  LDA #$00 ;LOW BYTE IS $00
1160  JSR WRITE
1170  LDX #31 ;DATA REGISTER
1180  LDY #0
1190 LOOP LDA SHAPE,Y
1200  JSR WRITE
1210  JSR WRITE
1220  INY
1230  CPY #8
1240  BCC LOOP
1250  RTS
1260 ;
1270 WRITE STX VDC
1280 WAIT BIT VDC
1290  BPL WAIT
1300  STA VDC+1
1310 RTS
```

Reading the 8563 Register

Reading a VDC 8563 register is exactly like writing to one—except that one step in the operation is carried out in reverse. In a VDC read operation, as in a write operation, the number of the VDC register to be accessed must be stored in memory address $D600. Then, just as in a write operation, a wait loop and a BIT instruction can be used to await the outcome of the 8563's work.

In a read operation, however, it is the 8563 chip—not the program being executed—that places a value in $D601. And that value is the value of the desired register. So, when the wait loop ends, the calling program can read the value of the chosen register by accessing memory address $D601. (A VDC read operation is in lines 1680 through 1720 of a program titled MOUSE80.SRC, which is presented later in this chapter.)

Giant Characters in 80 Columns

Now that you know how the 8563 chip works, we're ready to do some programming. The program we'll tackle is one called TALLCHRS.SRC, which appears in listing 17-2.

Listing 17-2 TALLCHRS.SRC program

```
1000 ;
1010 ; TALLCHRS.SRC
1020 ;
1030  *=$0D00
1040 ;
1050 MVSRCE = $FA
```

Listing 17-2 cont.

```
1060 LENPTR = MVSRCE+2
1070 TABLEN = $800
1080 REGNR = $0C00
1090 VDC = $D600
1100 CHRBAS = $D000
1110 INDFET = $FF74
1120 ;
1130 ; SET POINTERS
1140 ;
1150  LDA #<CHRBAS
1160  STA MVSRCE
1170  LDS #>CHRBAS
1180  STA MVSRCE+1
1190 ;
1200  LDA #<TABLEN
1210  STA LENPTR
1220  LDA #>TABLEN
1230  STA LENPTR+1
1240 ;
1250 ; CONFIGURE VDC CHIP FOR TALL CHRS
1260 ;
1270  LDX #9 ;CHAR SCAN LINE REGISTER
1280  LDA #15 ;NR OF SCAN LINES PER CHAR -1
1290  JSR WRITE
1300 ;
1310  LDX #6 ;LINE COUNT REGISTER
1320  LDA #12 ;12 LINES ON SCREEN
1330  JSR WRITE
1340 ;
1350  LDX #4 ;VERTICAL SCAN LINE REGISTER
1360  LDA #15 ;SETTING DEPENDS ON MONITOR
1370  JSR WRITE
1380 ;
1390  LDX #5 ;VERTICAL ADJUST REGISTER
1400  LDA #6 ;FINE TUNING FOR REGISTER 4
1410  JSR WRITE
1420 ;
1430  LDX #7 ;VERT SYNC REGISTER
1440  LDA #15 ;DEFINE UPPER BORDER OF
      SCREEN
1450  JSR WRITE
1460 ;
1470  LDX #23 ;CHAR HEIGHT REGISTER
1480  LDA #16 ;NR OF SCAN LINES IN CHAR
1490  JSR WRITE
1500 ;
1510 ; TELL VDC WHERE TO PUT NEW CHARACTERS
1520 ;
1530  LDX #18 ;ADDRESS UPDATE REGISTER,
      HIGH BYTE
```

Listing 17-2 cont.

```
1540  LDA #$20 ;HIGH BYTE OF ADDRESS TO BE
      CHANGED IS $20
1550  JSR WRITE
1560  INX ;NOW THE VALUE OF X IS #19
      (ADDRESS UPDATE REGISTER, LOW BYTE)
1570  LDA #$00 ;LOW BYTE IS $00
1580  JSR WRITE
1590 ;
1600 ; COPY ROM CHRS TO VDC RAM
1610 ;
1620  LDY #0
1630  LDX LENPTR+1
1640  BEQ MVPART
1650 MVPAGE JSR GETDATA
1660  INY
1670  BNE MVPAGE
1680  INC MVSRCE+1
1690  DEX
1700  BNE MVPAGE
1710 MVPART LDX LENPTR
1720  BEQ MVEXIT
1730 MVLAST JSR GETDATA
1740  INY
1750  DEX
1760  BNE MVLAST
1770 MVEXIT RTS
1780 ;
1790 GETDATA PHA
1800  TXA
1810  PHA
1820  LDA #MVSRCE ;POINTER TO CHRBAS
1830  LDX #14 ;BANK 14
1840  JSR INDFET ;DATA IS RETURNED IN
      ACCUMULATOR
1850  LDX #31 ;DATA REGISTER
1860  JSR WRITE
1870  JSR WRITE ;BLOW UP CHAR BY COPYING
      EACH BYTE TWICE
1880  PLA
1890  TAX
1900  PLA
1910  RTS
1920 ;
1930 WRITE STX VDC
1940 WAIT BIT VDC
1950  BPL WAIT
1960  STA VDC+1
1970  RTS
```

The TALLCHRS program—as you'll see when you type, assemble, and run it—can display double-height characters on the Commodore 128's 80-column screen. It is designed to be used with two other programs: CURSOR, which we saw back in listing 17-1, and TALLCHRS.BAS, a BASIC driver for the TALLCHRS and CURSOR programs. Because these three programs are designed to be used together, both assembly language programs must be assembled and all three programs must be stored on the same disk before the TALLCHRS program can be run. The TALLCHRS.BAS program appears in listing 17-3.

Listing 17-3
TALLCHRS.BAS
program

```
10 REM *** TALLCHRS.BAS ***
20 :
30 IF X=0 THEN X=1:BLOAD "TALLCHRS.OBJ"
40 SYS DEC("0D00")
50 IF XX=0 THEN XX=1:BLOAD "CURSOR.OBJ"
60 SYS DEC("0D00")
70 PRINT "{CLR}";:REM CLEAR SCREEN
80 PRINT "@";:GETKEY A$:PRINT CHR$(20);:
   PRINTA$;:REM PRINT CURSOR, BACK UP, GET
   TYPED CHARACTER, AND DISPLAY IT
90 GOTO 80:REM DO IT AGAIN
```

How the TALLCHRS.SRC Program Works

The TALLCHRS.SRC program is based on an odd but useful characteristic of the 8563 chip. As we saw at the beginning of this chapter, when the C-128 is placed in 80-column mode, it copies its built-in character set into the 8563's memory. But there is a difference between the way the C-128 character set is stored in ROM and the way that it is stored in the 8563's RAM. In the C-128's character generator ROM, each character is stored as a sequence of 8 bytes. Then, when a character is displayed on the screen, the 8 bytes that make up the character are stacked on top of each other. In this way, a display made up of 8-by-8 bit characters can be shown on the screen.

When characters from the 8563's RAM are displayed on the screen, the process is slightly different. In the VDC's RAM, each character is stored as a sequence of 16 bytes—twice the number of bytes allotted to each character in the C-128's character generator ROM. But under ordinary circumstances, only 8 of the 16 bytes stored in each character's RAM space are displayed on the screen.

The C-128 has two character sets, so this arrangement works very well. Because the VDC chip allocates 16 bytes to each character stored in RAM, two characters—a standard character and an alternate character—can be stored together in the space allocated to each individual character's space in the 8563's RAM. And, thanks to the character attribute system used by the 8563 chip, both sets of characters that can be stored in the chip's memory can be displayed on the screen at the same time.

Character Attribute System

The VDC's character attribute system works much like the VIC-II chip's color map system for generating screen colors. As we saw at the beginning of this chapter, the 8563 chip has a block of character attribute bytes that ordinarily extends from $0800 to $0FFF in the chip's 16K of internal RAM. On the 80-column screen map generated by the 8563 chip, each byte of data in the chip's character attribute block corresponds to the space occupied by one character on the screen. The codes stored in the 8563 attribute block can thus be used to control the color, appearance, and features of each character on the screen.

Because each byte in the VDC's attribute block contains eight bits, each attribute byte can be used to control eight features of its corresponding character on the screen. The eight bits in each attribute byte, and the character attribute controlled by each bit, are listed in table 17-3.

Table 17-3
Attributes and Their Corresponding Bits in 8563 RAM

Bit	Attribute
7	Alternate character set
6	Reverse video
5	Underline
4	Flashing character
3	Red*
2	Green*
1	Blue*
0	Intensity

*These three color bits and bit 0, the color intensity bit, can be combined to produce a total of 16 colors. Colors produced by bits 0 through 3 of each attribute bit—as well as by bits 0 through 3 and bits 4 through 7 of register 26—are illustrated in table 17-4.

Table 17-4
Colors Produced by the 8563's Color Attribute Bits

Bit Combination (in hexadecimal)	Color
0	Black
1	Dark gray
2	Blue
3	Light blue
4	Green
5	Light green
6	Cyan
7	Light cyan
8	Red
9	Light red
A	Purple
B	Light purple
C	Brown
D	Yellow
E	Light gray
F	White

Using Attributes and Registers Together

By using the VDC's attribute block and internal registers together, the chip can be programmed in many interesting ways. For example, characters of many different sizes, shapes, and colors can be displayed on the screen.

In the TALLCHRS.SRC program, the C-128's character set is copied into 8563 RAM as a set of double-height characters, each filling the entire 16 bytes allocated to every character in the VDC's memory. This trick takes place in lines 1930 and 1940, in which each byte of each character is merely copied twice into RAM.

When the C-128's character set has been copied into VDC RAM —as a set of giant-sized characters—various VDC registers that control the size of screen characters are set to display double-height characters on the screen.

This is where the TALLCHRS.BAS and CURSOR.SRC programs come in. First, TALLCHRS.BAS calls and executes the machine language version of TALLCHRS.SRC, which has been assembled and stored on a disk as TALLCHRS.OBJ. Then TALLCHRS.BAS calls and executes CURSOR.OBJ, the object code version of CURSOR.SRC. CURSOR.SRC turns the @ character in the 8563's RAM into an underline cursor —a task that isn't really very difficult, because the 8563 character set resides in RAM rather than ROM. Finally, when all of this is accomplished, the TALLCHRS.BAS program allows the user to type headline-sized characters on the C-128 screen.

Double High-Resolution Graphics

To take the 8563 out of its text mode and place it in its double high-resolution (bit-mapped) mode, all a programmer has to do is store an appropriate value in VDC register 25. If bit 7 of register 25 is cleared to 0, the VDC's text mode is enabled. By setting bit 7 to 1, a program can deactivate the chip's text mode and turn on its bit-mapped mode.

Under most circumstances, bit 6 of register 25 must also be reset to generate a bit-mapped display. Bit 6 activates and deactivates the VDC's character attribute RAM, and when bit-mapped mode is enabled, the 8563's attribute table usually must be disabled. When bit 6 of register 25 is set, screen attributes are enabled, and when bit 6 is clear, screen attributes are disabled. So bit 6 is usually cleared when the 8563's bit-mapped mode is used.

The reason that attribute RAM is disabled in bit-mapped mode is simply that there is not enough room for it in the 8563's RAM. When a bit-mapped screen is set up, it consumes most of the 8563 chip's 16K of RAM, leaving insufficient memory space for an attribute table. That is why only two colors are ordinarily available in the 8563's bit-mapped mode. There is one way to place more than two colors on the screen in bit-mapped mode; a program can reduce the size of the bit-mapped screen, leaving enough room in memory for an attribute table. But, unless you use a small screen for some special application, you probably won't have much use for this procedure.

When the VDC's attribute table is disabled, the color of each dot on the screen is controlled by the setting of VDC register 26; bits 0 through 3 of register 26 determine the color of each bit in screen memory that is turned on, and bits 4 through 7 determine the color of each bit in screen memory that is off. The colors produced by bits 0 through 3 and bits 4 through 7 of register 26 are listed in tables 17-3 and 17-4.

Plotting a Dot on a Double High-Resolution Screen

To bit map the 8563's screen, it is important to know what kind of system the VDC uses to translate bytes stored in a screen map to dots on a screen display. Fortunately, the 8563 plots the dots on its bit-mapped screen using a much simpler algorithm than the C-128'S VIC-II chip uses to generate its 40-column bit-mapped display. On the 8563's bit map, the bytes used to create a screen display are simply laid out in sequential order—beginning with memory address $0000 and ending with memory address $3E7F. So, although we must use a few algorithms to plot a dot on the 8563's screen, these algorithms are much simpler than the complex set of formulas used to plot a dot on the VIC-II's screen.

To place a dot on the 8563's screen, all a programmer has to do is assign the dot an X coordinate and a Y coordinate and then plot it on the screen using:

BYTE = INT(X/8) + Y × 80
BIT = 2↑(7 − (X AND 7))

MOUSE80.SRC Program

Our last program in this chapter is MOUSE80.SRC, a program that sets up an 80-column bit-mapped screen and then allows the C-128 user to draw on the screen with a hand controller, such as a mouse or a joystick. Along with the techniques of 8563 programming outlined in this chapter, it also makes use of the screen-drawing techniques that were used in the JOYSTICK program in a previous chapter. It is designed to be executed with the help of a short BASIC program titled MOUSE.BAS, which is shown in listing 17-4.

Listing 17-4
MOUSE.BAS
program

```
10 PRINT "{CLR}"
20 IF A=0 THEN A=1:BLOAD "MOUSE80.OBJ"
30 SYS DEC("1300")
```

The MOUSE80.SRC program took some time and effort to write, but it was well worth it—and I think you'll find the program well worth the time and effort it will take you to type, assemble, and study it. Have fun!

Listing 17-5
MOUSE80.SRC
program

```
1000 ;
1010 ; MOUSE80.SRC
1020 ;
1030  *=$1300
1040 ;
1050  TEMPTR = $C3
1060  JSV = $C8
1070  STATUS = TEMPTR+2
1080  HPSN = $FA
1090  VPSN = HPSN+2
1100  TEMPA = VPSN+1
1110  TEMPB = TEMPA+1
1120  DLCHR = $FF62
1130  VDC = $D600
1140  CIAPRA = $DC00
1150  INDFET = $FF74
1160 ;
1170  HMID = 316
1180  VMID =  90
1190  HMAX = 630
1200  VMAX = 180
1210  XMAX = HMAX+1
1220  YMAX = VMAX+1
1230 ;
1240  JMP START
1250 ;
1260 SETPTS .BYT $80,$40,$20,$10,$08,$04,
     $02,$01
1270 CLRPTS .BYT $7F,$BF,$DF,$EF,$F7,$FB,
     $FD,$FE
1280 ;
1290 ; MAIN MODULE
1300 ;
1310 START JSR HIRES ;ENTER HI-RES MODE
1320  JSR FIRSTDOT ;PLOT DOT AT MIDSCREEN
1330  JMP READJS
1340 ;
1350 PLOTDOT LDA HPSN
1360  PHA
1370  LDA HPSN+1
1380  PHA
1390  LDA VPSN
1400  PHA
1410  TXA
1420  PHA
1430  TYA
```

Listing 17-5 cont.

```
1440  PHA
1450 ;
1460  JSR PLOT
1470 ;
1480  PLA
1490  TAY
1500  PLA
1510  TAX
1520  PLA
1530  STA VPSN
1540  PLA
1550  STA HPSN+1
1560  PLA
1570  STA HPSN
1580  RTS
1590 ;
1600 ;VDC READ/WRITE ROUTINES
1610 ;
1620 WRITE STX VDC ;STORE IN VDC REGISTER
1630 WRWAIT BIT VDC ;TEST STATUS
1640  BPL WRWAIT ;NOT FINISHED YET
1650  STA VDC+1 ;STORE VALUE
1660  RTS
1670 ;
1680 READ STX VDC
1690 RDWAIT BIT VDC
1700  BPL RDWAIT
1710  LDA VDC+1
1720  RTS
1730 ;
1740 ;ENABLE HI-RES GRAPHICS
1750 ;
1760 HIRES LDX #25 ;SCROLL/MODE REGISTER
1770  LDA #$80 ;BIT 7 ENABLES HI-RES GRAPHICS
1780  JSR WRITE
1790 ;
1800 ;CLEAR HI-RES SCREEN
1810 ;
1820 CLRSCR LDY #64 ;CLEAR 64 PAGES
1830 LOOP LDX #18 ;ADR UPDATE REG
1840  TYA
1850  JSR WRITE
1860  LDX #31 ;DATA REGISTER
1870  LDA #$00
1880  JSR WRITE
1890  LDX #30 ;0 IN REG 31 MEANS...
1900  JSR WRITE ;...WRITE 256 CHARS
1910  DEY
1920  BPL LOOP ;UNTIL 64 PAGES CLEAR
```

Listing 17-5 cont.

```
1930  RTS
1940 ;
1950 ;PLOT DOT ON SCREEN
1960 ;
1970 ;DIVIDE X COORD BY 8
1980 ;
1990 PLOT LDA HPSN ;GET LOW BYTE OF X COORD
2000  STA TEMPB ;...& TUCK IT AWAY IN TEMPB
2010 ;
2020 ;3 SHIFTS TO RT = DIVISION BY 8
2030 ;
2040  LSR HPSN+1
2050  ROR HPSN
2060  LSR HPSN+1
2070  ROR HPSN
2080  LSR HPSN+1
2090  ROR HPSN
2100 ;
2110 ;MULTIPLY Y COORD BY 80
2120 ;(USING A COUPLE OF SHORTCUTS)
2130 ;
2140  LDA #$00 ;INITIALIZE HIGH BYTE
2150  STA TEMPA ;...TO 0
2160  LDA VPSN ;MULTIPLY Y COORD
2170  ASL VPSN ;... BY 2
2180  ROL TEMPA ;PICK UP CARRY
2190  ASL VPSN ;MULT BY 2 AGAIN
2200  ROL TEMPA ;PICK UP CARRY AGAIN
2210  ADC VPSN ;ADD RESULT TO ORIG VPSN
2220  STA VPSN ;PUT SUM IN VPSN
2230  BCC SKIP ;IF NO CARRY
2240  INC TEMPA ;ELSE PICK UP CARRY
2250 SKIP LDX #$04 ;SET UP 4X LOOP
2260 AGAIN ASL VPSN ;MULT BY 2
2270  ROL TEMPA ;PICK UP CARRY
2280  DEX ;DOWN TO 0 YET?
2290  BNE AGAIN ;IF NOT, LOOP BACK
2300 ;
2310 ; ADD Y*80 AND X/8
2320 ;
2330  LDA HPSN ;WHICH IS NOW X/8
2340  ADC VPSN ;WHICH IS NOW LOW BYTE OF
     (X/8+Y*80)
2350  STA VPSN ;PLACE LOW BYTE OF SUM IN
     VPSN
2360  BCC HOP ;IF NO CARRY
2370  INC TEMPA ;ELSE PICK UP CARRY
2380 ;
2390 ; BIT = 2↑(7-(X AND 7))
```

Listing 17-5 cont.

```
2400 ;
2410 HOP LDX #18 ;UPDATE REG HI BYTE
2420  LDA TEMPA ;HI BYTE OF (X/8+Y*80)
2430  JSR WRITE
2440  INX ;UPDATE REG LOW BYTE
2450  LDA VPSN ;LOW BYTE OF (X/8+Y*80)
2460  JSR WRITE
2470 ;
2480 ; STORE SUM IN UPDATE ADDRESS REGISTER
2490 ;
2500  LDX #31 ;DATA REGISTER
2510  JSR READ ;READ DATA REGISTER
2520  PHA ;SAVE LOW BYTE OF (X/8+Y*80)
2530  LDA TEMPB ;ORIG XPSN, LOW BYTE
2540  AND #$07 ;LIKE THE FORMULA SAYS
2550  TAX ;SAVE RESULT IN X
2560  PLA ;RETRIEVE LOW BYTE OF (X/8+Y*80)
2570  ORA SETPTS,X ;GET SETPTS DATA
2580  PHA ;SAVE MODIFIED BYTE
2590 ;
2600 ; PLACE NEW BYTE IN (X/8+Y*80)
2610 ;
2620  LDX #18 ;UPDATE REG HI BYTE
2630  LDA TEMPA ;HI BYTE OF ADDRESS
2640  JSR WRITE
2650  INX ;UPDATE REG LOW BYTE
2660  LDA VPSN ;LO BYTE OF ADDRESS
2670  JSR WRITE
2680  LDX #31 ;DATA REGISTER
2690  PLA ;RETRIEVE MODIFIED BYTE
2700  JSR WRITE
2710  LDX #18 ;UPDATE REG HI BYTE
2720  JSR READ ;...TO END THIS OPERATION
2730  RTS
2740 ;
2750 ; ENTER TEXT MODE
2760 ;
2770 TEXTON LDX #25 ;REGISTER 25
2780  LDA #$40 ;SET ATTRIBUTE BIT AND
2790  JSR WRITE ;...ENABLE TEXT MODE
2800  JSR DLCHR ;COPY ROM CHRS TO RAM
2810  RTS
2820 ;
2830 ; CLEAR CARRY BIT
2840 ;
2850 CLRPNT CLC ;CLR CARRY TO ERASE POINT
2860  JSR PLOT ;PLOT DOT ON SCREEN
2870  RTS
2880 ;
```

Listing 17-5 cont.

```
2890 ; PLOT DOT AT MIDSCREEN
2900 ;
2910 FIRSTDOT LDA #VMID
2920  STA VPSN
2930  LDA #<HMID
2940  STA HPSN
2950  LDA #>HMID
2960  STA HPSN+1
2970  JSR PLOTDOT
2980  RTS
2990 ;
3000 ; GET CONTENTS OF CIAPRA REG.
3010 ;
3020 GETCIA LDA #<CIAPRA
3030  STA TEMPTR
3040  LDA #>CIAPRA
3050  STA TEMPTR+1
3060  LDA #TEMPTR
3070  LDX #15 ;BANK 15
3080  LDY #0 ;INDEX 0
3090  JSR INDFET ;KERNEL ROUTINE
3100  STA STATUS
3110  RTS
3120 ;
3130 ; READ JOYSTICK
3140 ;
3150 READJS JSR GETCIA
3160  AND #$10 ; TRIGGER PRESSED?
3170  BNE CONTINUE; NO, CONTINUE
3180  JMP START ;YES, START OVER
3190 CONTINUE LDA #$0F
3200  PHA
3210  AND STATUS
3220  STA JSV
3230  PLA
3240  SEC
3250  SBC JSV
3260  STA JSV
3270 ;
3280  TAX
3290  BEQ READJS
3300  LDA RELADS-1,X
3310  STA MODREL+1
3320 MODREL BNE *
3330  MODR1
3340 ;
3350 ; ROUTINES TO MOVE JOYSTICK
3360 ;
3370 UP JSR MOVEUP
```

Listing 17-5 cont.

```
3380  JMP DONE
3390 ;
3400 DOWN JSR MOVEDN
3410  JMP DONE
3420 ;
3430 DNANDL JSR MOVEDN
3440  JMP LEFT
3450 UPANDL JSR MOVEUP
3460 LEFT LDX HPSN
3470  LDY HPSN+1
3480  TXA
3490  BNE DECLSB
3500  DEY
3510 DECLSB DEX
3520  STX HPSN
3530  STY HPSN+1
3540  JMP DONE
3550 ;
3560 DNANDR JSR MOVEDN
3570  JMP RIGHT
3580 UPANDR JSR MOVEUP
3590 RIGHT LDX HPSN
3600  LDY HPSN+1
3610  INX
3620  BNE NOINC
3630  INY
3640 NOINC STX
3650  STY HPSN+1
3660  JMP DONE
3670 ;
3680 ; SUBROUTINES TO MOVE UP & DOWN
3690 ;
3700 MOVEUP DEC VPSN
3710  RTS
3720 ;
3730 MOVEDN INC VPSN
3740  RTS
3750 ;
3760 DONE JSR CHECK
3770  JSR PLOTDOT
3780 ;
3790 ; A SHORT DELAY...
3800 ;
3810  LDX #255
3820 DLOOP NOP
3830  DEX
3840  BNE DLOOP
3850  JMP READJS
3860 ;
```

Listing 17-5 cont.

```
3870 ; MAKE SURE DOT IS WITHIN RANGE
3880 ;
3890 CHECK LDA VPSN
3900  BEQ RAISE
3910  CMP #VMAX-1
3920  BCS LOWER
3930  JMP HCHECK
3940 RAISE INC VPSN
3950  JMP HCHECK
3960 LOWER LDA #VMAX-1
3970  STA VPSN
3980 ;
3990 HCHECK BIT HPSN+1
4000  BPL OKLOW
4010  LDA #1
4020  STA HPSN
4030  LDA #0
4040  STA HPSN+1
4050  RTS
4060 ;
4070 OKLOW LDA #<XMAX
4080  CMP HPSN
4090  LDA #>XMAX
4100  SBC HPSN+1
4110  BCC TOOHI
4120  RTS
4130 ;
4140 TOOHI LDA #<HMAX
4150  STA HPSN
4160  LDA #>HMAX
4170  STA HPSN+1
4180  RTS
4190 ;
4200 RELADS .BYT UP-MODR1
4210  .BYT DOWN-MODR1
4220  .BYT READJS-MODR1
4230  .BYT LEFT-MODR1
4240  .BYT UPANDL-MODR1
4250  .BYT DNANDL-MODR1
4260  .BYT READJS-MODR1
4270  .BYT RIGHT-MODR1
4280  .BYT UPANDR-MODR1
4290  .BYT DNANDR-MODR1
4300 ;
```

A

6502/6510/8502 Instruction Set

This appendix is a complete listing of the 6502/6510/8502 microprocessor instruction set—all of the instruction mnemonics used in Commodore 64/128 assembly language programming. It does not include pseudo operations (also known as pseudo ops, or directives), which vary from assembler to assembler. Following are the abbreviations used in this appendix.

Abbreviations

Processor Status (P) Register Flags

N	Negative (sign) flag
V	Overflow flag
B	Break flag
D	Decimal flag
I	Interrupt flag
Z	Zero flag
C	Carry flag

6502/6510/8502 Memory Registers

A	Accumulator
X	X register
Y	Y register
M	Memory register

Addressing Modes

A	Absolute addressing
AC	Accumulator addressing
Z	Zero Page addressing
IMM	Immediate addressing
IND	Indirect addressing
IMP	Implicit (Implied) addressing
AX	Absolute Indexed,X addressing
AY	Absolute Indexed,Y addressing
IX	Indexed Indirect addressing
IY	Indirect Indexed addressing
R	Relative addressing
ZX	Zero Page,X addressing
ZY	Zero Page,Y addressing

6502/6510/8502 Instruction Set (Mnemonics)

ADC Add with carry.

Adds the contents of the accumulator to the contents of a specified memory location or literal value. If the P register's carry flag is set, a carry is also added. The result of the addition operation is then stored in the accumulator.

Flags affected: N, V, Z, C
Register affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

AND Logical AND.
Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location or an immediate value. The result of the operation is stored in the accumulator.

Flags affected: N, Z
Register affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

ASL Arithmetic shift left.
Moves each bit in the accumulator or a specified memory location one position to the left. A zero is placed into the bit 0 position, and bit 7 is moved into the carry bit of the P register. The result of the operation is left in the accumulator or the affected memory register.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

BCC Branch if carry clear.
Executes a branch if the carry flag is clear. There is no operation if the carry flag is set. Destination of branch must be within a range of −128 to +128 memory addresses from the BCC instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BCS Branch if carry set.
Executes a branch if the carry flag is set. There is no operation if the carry flag is clear. Destination of branch must be within a range of −128 to +128 memory addresses from the BCS instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BEQ Branch if equal.
Executes a branch if the zero flag is set. There is no operation if the zero flag is clear. Can be used to jump to cause a branch if the result of a calculation is zero, or if two numbers are equal. Destination of branch must be within a range of −128 to +128 memory addresses from the BEQ instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BIT Compare bits in accumulator with bits in a specified memory register.
Performs a binary logical AND operation on the contents of the

accumulator and the contents of a specified memory location. The contents of the accumulator are not affected, but three flags in the P register are.

If any bits in the accumulator and the value being tested match, the Z flag is cleared. If no match is found, the Z flag is set. Therefore, a BIT instruction followed by a BNE instruction can detect a match, and a BIT instruction followed by a BEQ instruction can detect a no-match condition.

In addition, bits 6 and 7 of the value in memory being tested are transferred directly into the V and N bits of the status register. This feature of the BIT instruction is often used in signed binary arithmetic. If a BIT operation results in the setting of the N flag, then the value being tested is negative. If the operation results in the setting of the V flag, this indicates a carry in signed-number math.

Flags affected: N, V, Z
Registers affected: None
Addressing modes: A, Z

BMI Branch on minus.
Executes a branch if the N flag is set. There is no operation if the N flag is clear. Destination of branch must be within a range of −128 to +128 memory addresses from the BMI instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BNE Branch if not equal.
Executes a branch if the zero flag is clear (that is, if the result of an operation is not zero). There is no operation if the zero flag is set. Can be used to jump to cause a branch if the result of a calculation is not zero, or if two numbers are not equal. Destination of branch must be within a range of −128 to +128 memory addresses from the BEQ instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BPL Branch on plus.
Executes a branch if the N flag is clear (that is, if the result of a calculation is positive). There is no operation if the N flag is set. Destination of branch must be within a range of −128 to +128 memory addresses from the BMI instruction.

Flags affected: None
Registers affected: None
Addressing mode: R

BRK Break.
Halts the execution of a program, much like an interrupt. Also stores the value of the program counter, plus two, on the hardware stack, in addition to the contents of the P register (which now has the B flag set). BRK is often used in debugging, and affects debuggers in various

ways. For more details, see your assembler's and debugger's instruction manuals.

Flag affected: B
Registers affected: None
Addressing mode: IMP

BVC Branch if overflow clear.

Executes a branch if the P register's overflow (V) flag is clear. There is no operation if the overflow flag is set. This instruction is used primarily in operations involving signed numbers.

Flags affected: None
Registers affected: None
Addressing mode: R

BVS Branch if overflow set.

Executes a branch if the P register's overflow (V) flag is set. Results in no operation if the overflow flag is clear. This instruction is used primarily in operations involving signed numbers.

Flags affected: None
Registers affected: None
Addressing mode: R

CLC Clear carry.

Clears the carry bit of the processor status register.

Flag affected: C
Registers affected: None
Addressing mode: IMP

CLD Clear decimal mode.

Puts the computer into binary mode (its default mode) so that binary operations (the kind most often used) can be carried out properly.

Flag affected: D
Registers affected: None
Addressing mode: IMP

CLI Clear interrupt mask.

Enables interrupts. Used in advanced assembly language programming. For more details, see advanced Commodore assembly language texts and manuals.

Flag affected: I
Registers affected: None
Addressing mode: IMP

CLV Clear overflow flag.

Clears the P register's overflow flag by setting it to zero. This instruction is used primarily in operations involving signed numbers.

Flag affected: V
Registers affected: None
Addressing mode: IMP

CMP Compare with accumulator.

Compares a specified literal number, or the contents of a specified memory location, with the contents of the accumulator. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the accumulator is less than, equal to, or greater than the value being tested.

Flags affected: N, Z, C
Registers affected: None
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

CPX Compare with X register.
Compares a specified literal number, or the contents of a specified memory location, with the contents of the X register. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the X register is less than, equal to, or greater than the value being tested.

Flags affected: N, Z, C
Registers affected: None
Addressing modes: A, IMM, Z

CPY Compare with Y register.
Compares a specified literal number, or the contents of a specified memory location, with the contents of the Y register. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the Y register is less than, equal to, or greater than the value being tested.

Flags affected: N, Z, C
Registers affected: None
Addressing modes: A, IMM, Z

DEC Decrement a memory location.
Decrements the contents of a specified memory location by one. If the value in the location is $00, the result of a DEC operation is $FF because there is no carry.

Flags affected: N, Z
Register affected: M
Addressing modes: A, Z, AX, ZX

DEX Decrement X register.
Decrements the X register by one. If the value in the location is $00, the result of the DEX operation is $FF because there is no carry.

Flags affected: N, Z
Register affected: X
Addressing mode: IMP

DEY Decrement Y register.
Decrements the Y register by one. If the value in the location is $00, the result of the DEY operation is $FF because there is no carry.

Flags affected: N, Z
Register affected: Y
Addressing mode: IMP

EOR Exclusive OR with accumulator.
Performs an exclusive OR operation on the contents of the accumulator and a specified literal value or memory location. The N and Z flags are set or cleared in accordance with the result of the operation, and the result is stored in the accumulator.

Flags affected: N, Z
Register affected: A
Addressing modes: A, Z, I, AX, AY, IX, IY, ZX

INC Increment memory.
The contents of a specified memory location are incremented by one. If the value in the location is $FF, the result of the INC operation is $00 because there is no carry.

Flags affected: N, Z
Register affected: M
Addressing modes: A, Z, AX, ZX

INX Increment X register.
The contents of the X register are incremented by one. If the value of the X register is $FF, the result of the INX operation is $00 because there is no carry.

Flags affected: N, Z
Register affected: X
Addressing mode: IMP

INY Increment Y register.
The contents of the Y register are incremented by one. If the value of the Y register is $FF, the result of the INY operation is $00 because there is no carry.

Flags affected: N, Z
Register affected: X
Addressing mode: IMP

JMP Jump to address.
Causes program execution to jump to the specified address. The JMP instruction can be used with absolute addressing, and it is the only 6510 instruction that can be used with unindexed indirect addressing. A JMP instruction that uses indirect addressing is written in the format:

```
JMP ($0600)
```

If this statement is used in a program, the JMP instruction causes program execution to jump to the value stored in memory address $0600, not to that address.

Flags affected: None
Registers affected: None
Addressing modes: A, IND

JSR Jump to subroutine.
Causes program execution to jump to the address that follows the instruction. The address should be the starting address of a subroutine that ends with an RTS instruction. When the program reaches the RTS instruction, execution of the program returns to the next

instruction after the JSR instruction that caused the jump to the subroutine.

Flags affected: None
Registers affected: None
Addressing mode: A

LDA Load the accumulator.
Loads the accumulator with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the accumulator, and the Z flag is set if the value loaded into the accumulator is zero.

Flags affected: N, Z
Register affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

LDX Load the X register.
Loads the X register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the X register, and the Z flag is set if the value loaded into the X register is zero.

Flags affected: N, Z
Register affected: X
Addressing modes: A, Z, IMM, AY, ZY

LDY Load the Y register.
Loads the Y register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the Y register, and the Z flag is set if the value loaded into the Y register is zero.

Flags affected: N, Z
Register affected: Y
Addressing modes: A, Z, IMM, AX, ZX

LSR Logical shift right.
Moves each bit in the accumulator one position to the right. A zero is placed into the bit 7 position, and bit 0 is placed into the carry. The result is left in the accumulator or the affected memory register.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

NOP No operation.
Causes the computer to do nothing for one or more cycles. Used in delay loops and to synchronize the timing of computer operations.

Flags affected: None
Registers affected: None
Addressing mode: IMP

ORA Inclusive OR with the accumulator.
Performs a binary inclusive OR operation on the value in the accumulator and either a literal value or the contents of a specified memory location. The N and Z flags are set or cleared in accordance with

the result of the operation, and the result of the operation is deposited in the accumulator.

Flags affected: N, Z
Registers affected: A, M
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

PHA Push accumulator.
The contents of the accumulator are pushed on the stack. The accumulator and the P register are not affacted.

Flags affected: None
Registers affected: None
Addressing mode: IMP

PHP Push processor status.
The contents of the P register are pushed on the stack. The P register itself is left unchanged, and no other registers are affected.

Flags affected: None
Registers affected: None
Addressing mode: IMP

PLA Pull accumulator.
One byte is removed from the stack and deposited in the accumulator. The N and Z flags are conditioned, as if an LDA operation had been carried out.

Flags affected: N, Z
Register affected: A
Addressing mode: IMP

PLP Pull processor status.
One byte is removed from the stack and deposited in the P register. This instruction is used to retrieve the status of the P register after it has been saved by pushing it onto the stack. All of the flags are thus conditioned to reflect the original status of the P register.

Flags affected: N, V, B, D, I, Z, C
Registers affected: None
Addressing mode: IMP

ROL Rotate left.
Each bit in the accumulator or a specified memory location is moved one position to the left. The carry bit is placed into the bit 0 location, and is replaced by bit 7 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

ROR Rotate right.
Moves each bit in the accumulator or a specified memory location one position to the right. The carry bit is placed into the bit 7 location, and the carry bit is replaced by bit 0 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

RTI Return from interrupt.
The status of both the program counter and the P register are restored in preparation for resuming the routine that was in progress when an interrupt occurred. All flags of the P register are restored to their original values. Interrupts are used in advanced assembly language programs, and detailed information on interrupts is available in advanced assembly language texts and Commodore reference manuals.
Flags affected: N, V, B, D, I, Z, C
Registers affected: None
Addressing mode: IMP

RTS Return from subroutine.
At the end of a subroutine, returns execution of a program to the next address after the JSR instruction that caused the program to jump to the subroutine. At the end of an assembly language program, the RTS instruction returns control of the computer to the device that was in control before the program began—usually the screen editor.
Flags affected: None
Registers affected: None
Addressing mode: IMP

SBC Subtract with carry.
Subtracts a literal value or the contents of a specified memory location from the contents of the accumulator. The opposite of the carry is also subtracted—in other words, there is a borrow. The result of the operation is stored in the accumulator.
Flags affected: N, V, Z, C
Register affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

SEC Set carry.
The carry flag is set. This instruction usually precedes an SBC instruction. Its primary purpose is to set the carry flag so that there can be a borrow.
Flag affected: C
Registers affected: None
Addressing mode: IMP

SED Set decimal mode.
Prepares the computer for operations using BCD (binary coded decimal) numbers. Although BCD arithmetic is more accurate than binary arithmetic (the usual type of 6510 arithmetic), it is slower and more difficult to use and uses more memory. BCD arithmetic is usually used in accounting and bookkeeping programs and in floating-point arithmetic.
Flag affected: D
Registers affected: None
Addressing mode: IMP

SEI Set interrupt disable.
Disables the interrupt response to an IRQ (maskable interrupt). Does not disable the response to an NMI (non-maskable interrupt). Interrupts are used in advanced assembly language programming, and are described in advanced assembly language texts and Commodore reference manuals.

Flag affected: I
Registers affected: None
Addressing mode: IMP

STA Store accumulator.
Stores the contents of the accumulator in a specified memory location. The contents of the accumulator are not affected.

Flags affected: None
Register affected: M
Addressing modes: A, Z, AX, AY, IX, IY, ZX

STX Store X register.
Stores the contents of the X register in a specified memory location. The contents of the X register are not affected.

Flags affected: None
Register affected: M
Addressing modes: A, Z, ZY

STY Store Y register.
Stores the contents of the Y register in a specified memory location. The contents of the Y register are not affected.

Flags affected: None
Register affected: M
Addressing modes: A, Z, ZX

TAX Transfer accumulator to X register.
The value in the accumulator is stored in the X register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z
Register affected: X
Addressing mode: IMP

TAY Transfer accumulator to Y register.
The value in the accumulator is stored in the Y register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z
Register affected: X
Addressing mode: IMP

TSX Transfer stack to X register.
The value in the stack pointer is stored in the X register. The N and C flags are conditioned in accordance with the result of this operation. The value of the stack pointer is not changed.

Flags affected: N, C
Register affected: X

Addressing mode: IMP

TXA Transfer X register to accumulator.
The value in the X register is deposited in the accumulator. The N and Z flags are conditioned in accordance with the result of this operation. The value of the X register is not changed.

Flags affected: N, Z
Register affected: A
Addressing mode: IMP

TXS Transfer X register to stack.
The value in the X register is stored in the stack pointer. No flags are conditioned by this operation. The value of the X register is not changed.

Flags affected: None
Registers affected: None
Addressing mode: IMP

TYA Transfer Y register to accumulator.
The value in the Y register is stored in the accumulator. The N and Z flags are conditioned by this operation. The value of the Y register is not changed.

Flags affected: N, Z
Register affected: A
Addressing mode: IMP

B

Commodore 128 Kernel Routines

The routines listed in this appendix are built into the Commodore 128's ROM and can be easily incorporated into user-written programs. To use a kernel routine, all a program has to do is fulfill the listed conditions and then do a JSR to the routine's vector address. The vector addresses in this appendix are pointers to a jump table, or table of JSR and JMP statements, that appears at memory addresses $E000 through $FFFF in bank 15. Although the actual addresses of the kernel routines may change as the C-128 is updated and improved, their vector addresses are guaranteed not to change. So the following routines and vector addresses can be safely used in user-written programs.

Name	Vector Address	Description
ACPTR	$FFA5	Input a byte from the serial bus. A single byte is accepted from the serial bus and returned in the accumulator. The TALK call should be used to prepare for this routine. Most applications should use a higher-level I/O routine such as GETIN. Register changed: A.
BASIN	$FFCF	Input a byte from the input channel. A byte is fetched from the current input device and returned in the accumulator. The default device is the keyboard. CHKIN may be used prior to calling BASIN to change the input device. On the Commodore 64, this routine was called CHRIN but had the same call vector and worked the same way. Register changed: A.
BOOT CALL*	$FF53	Boot a program from disk. Loads and executes the desired boot sector from an auto-boot disk. Before calling this routine, load the accumulator with the ASCII value of the drive number, and load the X register with the device number of your disk drive. Then, when BOOT CALL is called, it will call and execute the boot sector of an auto-boot disk. If there is an error, the command UI is set to the disk drive and the boot operation is aborted. Registers changed: A, X, Y.
BSOUT	$FFD2	Output a byte to the output channel. Writes the character in the accumulator to the current output device, usually the screen. The output device can be changed by calling CKOUT prior to calling BSOUT. On the Commodore 64, this call was named CHROUT but was functionally identical. Register changed: A.
CHKIN	$FFC6	Set channel to input. Establishes an input channel using the device number in the X register. Before calling CHKIN, call OPEN and then load the X register with the logical file number corresponding to the channel to be

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		opened. CHKIN can then be called to set the channel to input. CHKIN should be called prior to any form of input except the keyboard. Registers changed: A, X, Y.
CINT	$FF81	Initialize screen editor. Initializes the 40-column and 80-column screen editors, performing all necessary initialization tasks except those that pertain to I/O operations. (I/O operations must be initialized using the IOINIT call.) Before calling CINT, interrupts should be disabled with an SEI instruction. After CINT is called, interrupts can be enabled using CLI. Registers changed: A, X, Y.
CIOUT	$FFA8	Output a byte to the serial bus. Places the value of the accumulator on the serial bus. Before calling this routine, call LISTN. Then, if a secondary address is required, call SECND. In most applications, BSOUT should be used instead of CIOUT. Register changed: A.
CKOUT	$FFC9	Set channel to output. Establishes an output channel using the device number in the X register. Before calling CKOUT, call OPEN and then load the X register with the logical file number corresponding to the channel to be opened. CKOUT can then be called to set the channel to output. CKOUT should be called prior to any form of output except the screen. In the Commodore 64, it was called CHKOUT but was functionally identical. Registers changed: A, X, Y.
CLALL	$FFE7	Closes all open files and channels. Caution: After writing to a file, do not use CLALL as a substitute for a CLOSE call; the result will be an unclosed file. Registers changed: A, X, Y.
CLOSE	$FFC3	Close a logical file. Closes the file whose number is in the accumulator. Registers changed: A, X, Y.
CLOSE ALL*	$FF4A	Close all files to a device. Closes all open I/O channels. If one of these is the current I/O channel, the default channel (keyboard or screen) is restored. Registers changed: A, X, Y.
CLRCH	$FFCC	Restore default channels. Closes all open channels and restores the keyboard and screen as input and output channels, respectively. Registers changed: A, X.
C64MODE*	$FF4D	Enter Commodore 64 mode. The Commodore 128 is placed in Commodore 64 mode. No unique C-128 features should be active when this call is issued. Registers changed: None.
DLCHR*	$FF62	Initialize 80-column characters. Copies the C-128's ROM character set into the 80-column 8563 VDC chip's character generator RAM block, filling the extra eight bytes in each 8563

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		character cell with zeros. Registers changed: A, X, Y.
DMA CALL*	$FF50	Send command to DMA device. Used to communicate with an expansion RAM cartridge. Registers changed: A, X.
GETCFG*	$FF6B	Get byte to configure MMU for any bank. Returns the MMU configuration code for the bank number (0 through 15) stored in the X register. The MMU code is returned in the accumulator. Register changed: A
GETIN	$FFE4	Get a byte from the input buffer. Reads a character from the current input device. The default input device is the keyboard. If another input device is to be read, the OPEN and CHKIN routines must be called before GETIN is called. Registers changed: A, X, Y.
INDCMP*	$FF7A	CMP (*nnnn*),Y to data in any bank, where *nnnn* is a zero page pointer. Compare the accumulator with a byte from any memory bank without leaving the currently active memory bank. To use the routine, do the following. Store the address of a zero page pointer in memory address $02C8, store the low byte and high byte of a base address in the pointer, load the X register with the number of the bank to be accessed (0 through 15), load the Y register with an index, and call INDCMP. The result of the comparison is returned in the processor status register. Registers affected: A, X.
INDFET*	$FF74	LDA (*nnnn*),Y from any bank, where *nnnn* is a zero page pointer. This routine loads a byte into the accumulator from any memory bank without leaving the bank that is currently active. To use the routine, load a base address into a 2-byte zero page pointer, load the accumulator with the address of the pointer, load the X register with the number of the bank to be accessed (0 through 15), load the Y register with an index, and call INDFET. Register affected: X.
INDSTA*	$FF77	STA (*nnnn*),Y in any bank, where *nnnn* is a zero page pointer. This routine stores a byte in any memory bank without leaving the bank that is currently active. To use the routine, do the following. Store the address of a zero page pointer in memory address $02C9, store the low byte and high byte of a base address in the pointer, load the accumulator with the byte to be stored, load the X address with the number of the bank to be accessed (0 through 15), load the Y register with an index, and call INDSTA. Register affected: X.
IOBASE	$FFF3	Get location of I/O block. Returns the low

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		byte of the I/O bank in the X register and the high byte in the Y register. This routine is not used by the Commodore 128 but was included in the C-128 kernel to enhance its compatibility with other Commodore computers. Registers affected: X, Y.
IOINIT	$FF84	Initialize I/O devices. Initializes all of the C-128's I/O chips and performs other initialization functions. Interrupts should be disabled before this routine is called. Registers affected: A, X, Y.
JMPFAR*	$FF71	JMP to any bank. To use this routine, store the number of the desired bank (0 through 15) in memory address $02, the high byte of the destination address in $04, the contents of the status register in $05, the contents of the accumulator in $06, the contents of the X register in $07, and the contents of the Y register in $08. Then call JMPFAR, and your program will jump to the desired address. Registers changed: None.
JSRFAR*	$FF6E	JSR to any bank. To use this routine, store the number of the desired bank (0 through 15) in memory address $02, the high byte of the destination address in $04, the contents of the status register in $05, the contents of the accumulator in $06, the contents of the X register in $07, and the contents of the Y register in $08. Then call JMPFAR, and your program will JSR to the desired address. On return, load the status register, accumulator, X register, and Y register with the values of memory addresses $05, $06, $07, and $08, respectively. Registers changed: None.
KEY	$FF9F	Read the keyboard. Scans the keyboard. If a key is pressed, its key code is placed in the keyboard buffer. This routine is called by the operating system 60 times per second, during the C-128's vertical blank interrupt. In the Commodore 64, it was called SCNKEY. Registers affected: None.
LISTN	$FFB1	Send listen command to serial device. Instructs an I/O device to start reading data. In most applications, a higher-level routine such as CKOUT should be used. Register affected: A.
LKUPLA*	$FF59	Look up logical file number in file tables. This routine is use primarily by BASIC DOS commands to find unused logical addresses when a disk channel needs to be established. To use this routine, load the Y register with the logical file number to be searched for and call LKUPLA. The file number, device number, and secondary address, if found, are

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		returned in the accumulator, the X register, and the Y register, respectively. Registers affected: A, X, Y.
LKUPSA*	$FF5C	Look up secondary address in file tables. This routine is used primarily by BASIC DOS commands to find unused logical addresses when a disk channel needs to be established. To use this routine, load the Y register with the secondary address to be searched for and call LKUPSA. The file number, device number, and secondary address, if found, are returned in the accumulator, the X register, and the Y register, respectively. Registers affected: A, X, Y.
LOAD	$FFD5	Load from file. Loads data from an input device into memory. Can also verify that data in memory matches that in a file. Prior to calling LOAD, call SETNAM, SETBNK, or SETLFS. Then load the accumulator with a zero to load, or a nonzero value to verify. If the secondary address specified by SETLFS is 0, load the X register with the low byte of the starting address, and the Y register with the high byte. If the secondary address is 1, the starting address is read from the device. Registers affected: A, X, Y.
MEMBOT	$FF9C	Set or read bottom of RAM. Can be used to read or set the bottom-of-memory pointer at $0A05 and $0A06. Because of the C-128's banked memory architecture, this routine has little use in C-128 programs. It was included in the C-128 kernel primarily for compatibility purposes. Registers affected: X, Y.
MEMTOP	$FF99	Set or read top of RAM. Can be used to read or set the top-of-memory pointer at $0A07 and $0A08. Because of the C-128's banked memory architecture, this routine has little use in C-128 programs. It was included in the C-128 kernel primarily for compatibility purposes. Registers affected: X, Y.
OPEN	$FFC0	Open a logical file. Prepares a logical file for I/O operations. Call SETNAM, SETBNK, and SETLFS before calling this routine. Registers affected: A, X, Y.
PFKEY*	$FF65	Program a function key. Replaces a string assigned to a function key with a user-written string. Prior to calling PFKEY, store the address of the new string (low byte, high byte, and bank number) in three successive zero page addresses. Then load the accumulator with the address of your 3-byte pointer, load the X register with the number of the function key to be reprogrammed, load the Y register

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		with the length of the new string, and call PFKEY. Registers affected: A, X, Y.
PHEONIX*	$FF56	Initialize function ROM cartridges. Locates all installed function ROM cartridges, calls them, looks for an auto-boot disk, and boots the disk if one is found. This routine is called by BASIC at the conclusion of a cold start. Registers affected: A, X, Y.
PLOT	$FFF0	Set or read cursor position. Sets or reads the position of the cursor in the currently open screen window. To set the cursor position, clear the carry bit, load the X register with the desired row number, load the Y register with the desired column number, and call PLOT. To read the cursor position, set the carry bit and call PLOT. The cursor's row number is returned in the X register, and the cursor's column number is returned in the Y register. Registers affected: X, Y.
PRIMM*	$FF7D	Print the following ASCII string. Prints an ASCII string on the current output device. The ASCII code for the string to be printed should immediately follow the PRIMM call. The string can contain up to 255 characters, and should end with a zero value. Registers affected: None.
RAMTAS	$FF87	Initialize RAM and buffers. Clears page zero RAM, initializes the cassette and serial buffers, initializes pointers to the top and bottom of RAM, and sets system vector addresses $0A00 and $0A01 to point to $4000, BASIC's cold-start address. Registers affected: A, X, Y.
RDTIM	$FFDE	Read jiffy clock. Returns the low, middle, and high bytes of the C-128's jiffy clock in the accumulator, X register, and Y register, respectively. Registers affected: A, X, Y.
READSS	$FFB7	Read I/O status. Returns, in the accumulator, a code number that reflects the outcome of the most recent I/O operation. Status bytes returned by the READSS call are the same as those returned by the ST command in BASIC. They are used primarily for error checking. Register affected: A.
RESTOR	$FF8A	Restore kernel indirect RAM vectors. Restores the default values of the kernel's indirect vectors. Interrupts should be disabled before this routine is called. Registers affected: A, X, Y.
SAVE	$FFD8	Save memory to a device. Saves data from memory to an output device. SETNAM, SETBNK, and SETLFS should be called before SAVE is called. Also, before SAVE is called, the starting address of the block to be saved should be placed in a zero page pointer, the

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		address of the pointer should be placed in the accumulator, and the address of the end of the block, plus one, should be stored in the X and Y registers. Registers affected: A, X, Y.
SCRORG	$FFED	Get size of current screen window. The width of the current screen window is returned in the X register, and the height of the register is returned in the Y register. Registers affected: A, X, Y.
SECND	$FF93	Send secondary address. Sends a secondary address to a device that has been initialized with a LISTN kernel call. In most applications, higher-level routines such as OPEN and CKOUT should be used instead of the SECND call. Register affected: A.
SETBNK*	$FF68	Set banks for I/O operations. This routine should be called prior to I/O operations such as OPEN, LOAD, and SAVE. Prior to calling SETNAM, load the accumulator with the desired bank number (0 through 15), and load the X register with the bank number containing the name of the file to be accessed. Registers affected: None.
SETLFS	$FFBA	Set logical file number, device number, and secondary address. Sets the logical file number, device number, and secondary address for higher-level kernel I/O routines. Prior to calling SETLFS, load the accumulator, the X register, and the Y register with the logical file number, device number, and secondary address, respectively. If there is no secondary address, the Y register can be loaded with the value $FF. Registers affected: None.
SETMSG	$FF90	Enable/disable kernel messages. Updates the kernel message flag byte ($9D) that determines whether terse kernel error messages or more verbose error messages will be displayed. If SETMSG is called with bit 6 of the accumulator set, kernel error messages are displayed. If the routine is called with bit 7 set, verbose error messages are displayed. Registers affected: None.
SETNAM	$FFBD	Set file name pointers. Sets up the file name or command string for higher-level kernel I/O calls such as OPEN, LOAD, and SAVE. Before SETNAM is called, the name of the file to be accessed should be stored in RAM as an ASCII string. Then the accumulator should be loaded with the length of the file name, the X and Y registers should be loaded with the address of the file name, and SETNAM should be called. If a file name is not needed, a name length of

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		zero can be specified. Registers affected: None.
SETTIM	$FFDB	Set jiffy clock. Sets the system clock with the low, middle, and high values stored in the accumulator, the X register, and the Y register, respectively. Registers affected: None.
SETTMO	$FFA2	Enable/disable IEEE timeouts. This routine is used by the C-64 to disable I/O timeouts that affect the IEEE communications cartridge. It is not used by the Commodore 128. Registers affected: None.
SPINP SPOUT*	$FF47	Set fast serial ports for input or output. SPINP and SPOUT are low-level routines that can be used to set up fast serial communications between the Commodore 128 and the 1571 disk drive. SPINP and SPOUT are not usually required in C-128 programs, because LOAD and SAVE commands automatically use fast serial I/O when communicating with the 1571 disk drive. Registers affected: A, X.
STOP	$FFE1	Read Run/Stop key. Checks to see whether the Run/Stop key is pressed. If it is, STOP calls CLRCH, clears the keyboard buffer, and returns with the Z flag set, enabling a program to branch to any desired address with a BEQ instruction. Registers affected: A, X.
SWAPPER*	$FF5F	Switch between 40 and 80 columns. Toggles the display mode between 40 and 80 columns. If 80-column mode is enabled, bit 7 of memory address $D7 is set. Registers affected: A, X, Y.
TALK	$FFB4	Send TALK command to serial device. Instructs a serial device to start sending data. Prior to calling TALK, the accumulator should be loaded with a device number ranging from 0 to 31. In most applications, a higher-level routine such as CHKIN should be used instead of the TALK routine. Register affected: A.
TKSA	$FF96	Send secondary address to a "talk" device. Sends a secondary address to a device that has been instructed, using a TALK call, to send data. In most applications, a higher-level routine such as OPEN or CHKIN should be used instead of the TKSA routine. Register affected: A.
UDTIM	$FFEA	Update jiffy clock. Increments the system software (jiffy) clock, a 3-byte timer at memory addresses $A0 through $A2. Also decrements a countdown clock called TIMER, which resides at memory addresses $A1D through $A1F. Registers affected: A, X.
UNLSN	$FFAE	Send "unlisten" command to serial device. Instructs all "listening" devices on the serial

* Commodore 128 only—not included in Commodore 64 kernel.

Name	Vector Address	Description
		bus to stop reading data. In most applications, a higher-level routine such as CLRCH should be used instead of the UNLSN routine. Register affected: A.
UNTLK	$FFAB	Send "untalk" command to serial device. Instructs all "talking" devices on the serial bus to stop sending data. In most applications, a higher-level routine such as CLRCH should be used instead of the UNTLK routine. Register affected: A.
VECTOR	$FF8D	Set or copy kernel indirect RAM vectors. This call can be used to replace the C-128's kernel vectors with a set of customized vectors. To copy the C-128's default kernel vectors into another section of RAM, set the carry bit and place the low and high bytes of the destination addresses in the X and Y registers, respectively. To replace the kernel vectors with user-written vectors, call the VECTOR routine with the carry bit set and with the address of the list of new vectors in the X and Y registers. Interrupts should be disabled before this routine is called. Registers affected: X, Y.

C

Commodore 128 Character Codes

The codes listed in this appendix are the values that the Commodore 128's operating system uses to print characters on the screen in response to a PRINT CHR$(X) statement or a BSOUT kernel call. The Commodore 128 actually has two sets of characters: one called the graphics/uppercase set and one called the uppercase/lowercase set. When the computer is turned on, the uppercase/graphics character set is active, but the computer can be switched to its lowercase/uppercase set with a PRINT CHR$(14) statement or an equivalent kernel call. To switch back to the uppercase/graphics character set, you can use a PRINT CHR$(142) statement or an equivalent kernel call.

The Commodore 128 also uses two other sets of character-related codes: a set of keyboard codes, which can be used to determine which key on the keyboard is pressed (or if no key is pressed), and a set of screen codes, which can be placed directly in screen memory to generate a screen display. The keyboard code set is listed in chapter 15, and the screen code set is listed in appendix D.

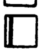
Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$00	0		
$01	1		
$02	2	Underline on (80 column)	Underline on (80 column)
$03	3	White	White
$04	4		
$05	5		
$06	6		
$07	7	Bell tone (C-128)	Bell tone (C-128)
$08	8	Disable Shift-C= (C-64)	Disable Shift-C= (C-64)
$09	9	Tab (C-128)	Tab (C-128)
		Enable Shift-C= (C-64)	Enable Shift-C= (C-64)
$0A	10	Line Feed (C-128)	Line Feed (C-128)
$0B	11	Disable Shift-C= (C-128)	Disable Shift-C= (C-128)
$0C	12	Enable Shift-C= (C-128)	Enable Shift-C= (C-128)
$0D	13	Return	Return
$0E	14	Switch to lowercase	Switch to lowercase
$0F	15	Flash on (80 column)	Flash on (80-column)
$10	16		
$11	17	Cursor down	Cursor down
$12	18	Reverse on	Reverse on
$13	19	Home	Home
$14	20	Delete	Delete
$15	21		

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$16	22		
$17	23		
$18	24	Tab set/clear (C-128)	Tab set/clear (C-128)
$19	25		
$1A	26		
$1B	27	Escape	Escape
$1C	28	Red	Red
$1D	29	Cursor right	Cursor right
$1E	30	Green	Green
$1F	31	Blue	Blue
$20	32	Space	Space
$21	33	!	!
$22	34	"	"
$23	35	#	#
$24	36	$	$
$25	37	%	%
$26	38	&	&
$27	39	'	'
$28	40	(	(
$29	41	)	)
$2A	42	*	*
$2B	43	+	+
$2C	44	,	,
$2D	45	-	-
$2E	46	.	.
$2F	47	/	/
$30	48	0	0
$31	49	1	1
$32	50	2	2
$33	51	3	3
$34	52	4	4
$35	53	5	5
$36	54	6	6
$37	55	7	7
$38	56	8	8
$39	57	9	9
$3A	58	:	:
$3B	59	;	;

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$3C	60	<	<
$3D	61	=	=
$3E	62	>	>
$3F	63	?	?
$40	64	@	@
$41	65	A	a
$42	66	B	b
$43	67	C	c
$44	68	D	d
$45	69	E	e
$46	70	F	f
$47	71	G	g
$48	72	H	h
$49	73	I	i
$4A	74	J	j
$4B	75	K	k
$4C	76	L	l
$4D	77	M	m
$4E	78	N	n
$4F	79	O	o
$50	80	P	p
$51	81	Q	q
$52	82	R	r
$53	83	S	s
$54	84	T	t
$55	85	U	u
$56	86	V	v
$57	87	W	w
$58	88	X	x
$59	89	Y	y
$5A	90	Z	z
$5B	91	[	[
$5C	92	£	£
$5D	93	]	]
$5E	94	Cursor up	Cursor up
$5F	95	Cursor left	Cursor left
$60	96	⊟	⊟
$61	97	♠	A

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$62	98	│	B
$63	99	─	C
$64	100	⎺	D
$65	101	▔	E
$66	102	⎽	F
$67	103	▏	G
$68	104	▕	H
$69	105	╮	I
$6A	106	╰	J
$6B	107	╯	K
$6C	108	⌞	L
$6D	109	╲	M
$6E	110	╱	N
$6F	111	⌜	O
$70	112	⌝	P
$71	113	●	Q
$72	114	▁	R
$73	115	♥	S
$74	116	▎	T
$75	117	╭	U
$76	118	╳	V
$77	119	○	W
$78	120	♣	X
$79	121	▕	Y
$7A	122	♦	Z
$7B	123	┼	┼
$7C	124	🮌	🮌
$7D	125	│	│
$7E	126	π	▒
$7F	127	◥	▧
$80	128		
$81	129	Orange (40 column) Dark purple (80 column)	Orange (40 column) Dark purple (80 column)
$82	130	Underline off (80 column)	80 column off (80 column)
$83	131		
$84	132	—	
$85	133	F1	F1
$86	134	F3	F3

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$87	135	F5	F5
$88	136	F7	F7
$89	137	F2	F2
$8A	138	F4	F4
$8B	139	F6	F6
$8C	140	F8	F8
$8D	141	Shift-Return	Shift-Return
$8E	142	Switch to uppercase	Switch to uppercase
$8F	143	Flash off (80 column)	Flash off (80 column)
$90	144	Black	Black
$91	145	Cursor up	Cursor up
$92	146	Reverse off	Reverse off
$93	147	Clear screen	Clear screen
$94	148	Insert	Insert
$95	149	Brown (40 column) Dark yellow (80 column)	Brown (40 column) Dark yellow (80 column)
$96	150	Light red	Light red
$97	151	Dark gray (40 column) Dark cyan (80 column)	Dark gray (40 column) Dark cyan (80 column)
$98	152	Medium gray	Medium gray
$99	153	Light green	Light green
$9A	154	Light blue	Light blue
$9B	155	Light gray	Light gray
$9C	156	Purple	Purple
$9D	157	Cursor left	Cursor left
$9E	158	Yellow	Yellow
$9F	159	Cyan	Cyan
$A0	160	Shift-Space	Shift-Space
$A1	161	◧	◧
$A2	162	⬓	⬓
$A3	163	□	□
$A4	164	□	▒
$A5	165	□	□
$A6	166	▒	□
$A7	167	□	□
$A8	168	⬓	□
$A9	169	◩	▨
$AA	170	◨	◨

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$AB	171		
$AC	172		
$AD	173		
$AE	174		
$AF	175		
$B0	176		
$B1	177		
$B2	178		
$B3	179		
$B4	180		
$B5	181		
$B6	182		
$B7	183		
$B8	184		
$B9	185		
$BA	186		
$BB	187		
$BC	188		
$BD	189		
$BE	190		
$BF	191		
$C0	192		
$C1	193		A
$C2	194		B
$C3	195		C
$C4	196		D
$C5	197		E
$C6	198		F
$C7	199		G
$C8	200		H
$C9	201		I
$CA	202		J
$CB	203		K
$CC	204		L
$CD	205		M
$CE	206		N
$CF	207		O
$D0	208		P

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$D1	209	●	Q
$D2	210	▁	R
$D3	211	♥	S
$D4	212	▏	T
$D5	213	╭	U
$D6	214	╳	V
$D7	215	○	W
$D8	216	♣	X
$D9	217	▕	Y
$DA	218	♦	Z
$DB	219	┼	┼
$DC	220	▌	▌
$DD	221	│	│
$DE	222	π	▒
$DF	223	◥	▧
$E0	224	Shift-Space	Shift-Space
$E1	225	▌	▌
$E2	226	▄	▄
$E3	227	▔	▔
$E4	228	▁	▒
$E5	229	▏	▏
$E6	230	▒	▁
$E7	231	▕	▕
$E8	232	▄	▎
$E9	233	◤	▨
$EA	234	▐	▐
$EB	235	├	├
$EC	236	▗	▗
$ED	237	└	└
$EE	238	┐	┐
$EF	239	▂	▂
$F0	240	┌	┌
$F1	241	┴	┴
$F2	242	┬	┬
$F3	243	┤	┤
$F4	244	▎	▎
$F5	245	▍	▍
$F6	246	▐	▐

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$F7	247	▔	▔
$F8	248	▀	▀
$F9	249	▃	▃
$FA	250		✓
$FB	251	▖	▖
$FC	252	▝	▝
$FD	253	┘	┘
$FE	254	▘	▘
$FF	255	π	▒

D

Commodore 128 Screen Codes

The codes in this appendix can be placed directly in the Commodore 128's 40-column or 80-column screen map to generate a screen display. Codes $80 through $FF (128 through 255 in decimal notation) are the reverse images of codes $00 through $7F (0 through 127 in decimal notation).

The Commodore 128 also uses two other sets of character-related codes: a set of keyboard codes, which can be used to determine which key on the keyboard is pressed (or if no key is pressed), and a set of character codes, which the operating system uses to print characters on the screen in response to PRINT CHR$(X) instructions and BSOUT kernel calls. The keyboard code set is listed in chapter 15, and the character code set is listed in appendix C.

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$00	0	@	@
$01	1	A	a
$02	2	B	b
$03	3	C	c
$04	4	D	d
$05	5	E	e
$06	6	F	f
$07	7	G	g
$08	8	H	h
$09	9	I	i
$0A	10	J	j
$0B	11	K	k
$0C	12	L	l
$0D	13	M	m
$0E	14	N	n
$0F	15	O	o
$10	16	P	p
$11	17	Q	q
$12	18	R	r
$13	19	S	s
$14	20	T	t
$15	21	U	u
$16	22	V	v
$17	23	W	w
$18	24	X	x
$19	25	Y	y
$1A	26	Z	z

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$1B	27	[	[
$1C	28	£	£
$1D	29	]	]
$1E	30	Cursor up	Cursor up
$1F	31	Cursor left	Cursor left
$20	32	Space	Space
$21	33	!	!
$22	34	"	"
$23	35	#	#
$24	36	$	$
$25	37	%	%
$26	38	&	&
$27	39	'	'
$28	40	(	(
$29	41	)	)
$2A	42	*	*
$2B	43	+	+
$2C	44	,	,
$2D	45	-	-
$2E	46	.	.
$2F	47	/	/
$30	48	0	0
$31	49	1	1
$32	50	2	2
$33	51	3	3
$34	52	4	4
$35	53	5	5
$36	54	6	6
$37	55	7	7
$38	56	8	8
$39	57	9	9
$3A	58	:	:
$3B	59	;	;
$3C	60	<	<
$3D	61	=	=
$3E	62	>	>
$3F	63	?	?
$40	64	⊟	⊟

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$41	65	♠	A
$42	66	│	B
$43	67	─	C
$44	68	─	D
$45	69	▔	E
$46	70	─	F
$47	71	▏	G
$48	72	▕	H
$49	73	╮	I
$4A	74	╰	J
$4B	75	╯	K
$4C	76	└	L
$4D	77	╲	M
$4E	78	╱	N
$4F	79	┌	O
$50	80	┐	P
$51	81	●	Q
$52	82	▁	R
$53	83	♥	S
$54	84	▏	T
$55	85	╭	U
$56	86	╳	V
$57	87	○	W
$58	88	♣	X
$59	89	▕	Y
$5A	90	♦	Z
$5B	91	┼	┼
$5C	92	▒	▒
$5D	93	│	│
$5E	94	π	▒
$5F	95	◥	▨
$60	96	Shift-Space	Shift-Space
$61	97	▌	▌
$62	98	▄	▄
$63	99	▔	▔
$64	100	▁	▒
$65	101	▏	▏
$66	102	▒	▔

Hex	Decimal	Uppercase/Graphics	Uppercase/Lowercase
$67	103	▕	▁
$68	104	🮏	▏
$69	105	◤	▨
$6A	106	🮇	🮇
$6B	107	├	├
$6C	108	▗	▗
$6D	109	└	└
$6E	110	┐	┐
$6F	111	▂	▂
$70	112	┌	┌
$71	113	┴	┴
$72	114	┬	┬
$73	115	┤	┤
$74	116	▎	▎
$75	117	▍	▍
$76	118	🮈	🮈
$77	119	🮂	🮂
$78	120	🮃	🮃
$79	121	▃	▃
$7A	122	⌟	✓
$7B	123	▖	▖
$7C	124	▝	▝
$7D	125	┘	┘
$7E	126	▘	▘
$7F	127	▚	▚

E

Commodore 128 Frequency Codes

This appendix lists the codes that can be placed in the 6581 SID chip's low and high frequency registers to produce the following notes. Instructions for using these codes in Commodore 128 programs are in chapter 15.

		Frequency	
Octave	**Pitch**	**High Byte**	**Low Byte**
0	C	1	12
	C#	1	28
	D	1	45
	D#	1	62
	E	1	81
	F	1	102
	F#	1	123
	G	1	145
	G#	1	169
	A	1	195
	A#	1	221
	B	1	250
1	C	2	24
	C#	2	56
	D	2	90
	D#	2	125
	E	2	163
	F	2	204
	F#	2	246
	G	3	35
	G#	3	83
	A	3	134
	A#	3	187
	B	3	244
2	C	4	48
	C#	4	112
	D	4	180
	D#	4	251
	E	5	71
	F	5	152
	F#	5	237
	G	6	71
	G#	6	167
	A	7	12
	A#	7	119
	B	7	233
3	C	8	97
	C#	8	225
	D	9	104
	D#	9	247
	E	10	143
	F	11	48
	F#	11	218
	G	12	143
	G#	13	78
	A	14	24
	A#	14	239
	B	15	210

		Frequency	
Octave	**Pitch**	**High Byte**	**Low Byte**
4	C	16	195
	C#	17	195
	D	18	209
	D#	19	239
	E	21	31
	F	22	96
	F#	23	181
	G	25	30
	G#	26	156
	A	28	49
	A#	29	223
	B	31	165
5	C	33	135
	C#	35	134
	D	37	162
	D#	39	223
	E	42	62
	F	44	193
	F#	47	107
	G	50	60
	G#	53	57
	A	56	99
	A#	59	190
	B	63	75
6	C	67	15
	C#	71	12
	D	75	69
	D#	79	191
	E	84	125
	F	89	131
	F#	94	214
	G	100	121
	G#	106	115
	A	112	199
	A#	119	124
	B	126	151
7	C	134	30
	C#	142	24
	D	150	139
	D#	159	126
	E	168	250
	F	179	6
	F#	189	172
	G	200	243
	G#	212	230
	A	225	143
	A#	238	284
	B	253	46

F

Commodore 128 Memory Maps

This appendix contains memory maps used by the Commodore 128 in all three of its operating modes: CP/M mode, C-64 mode, and the computer's native C-128 mode. Instructions for using these maps, and explanations of their contents, are in chapters 1 and 10.

Figure F-1
C-128's three blocks of memory

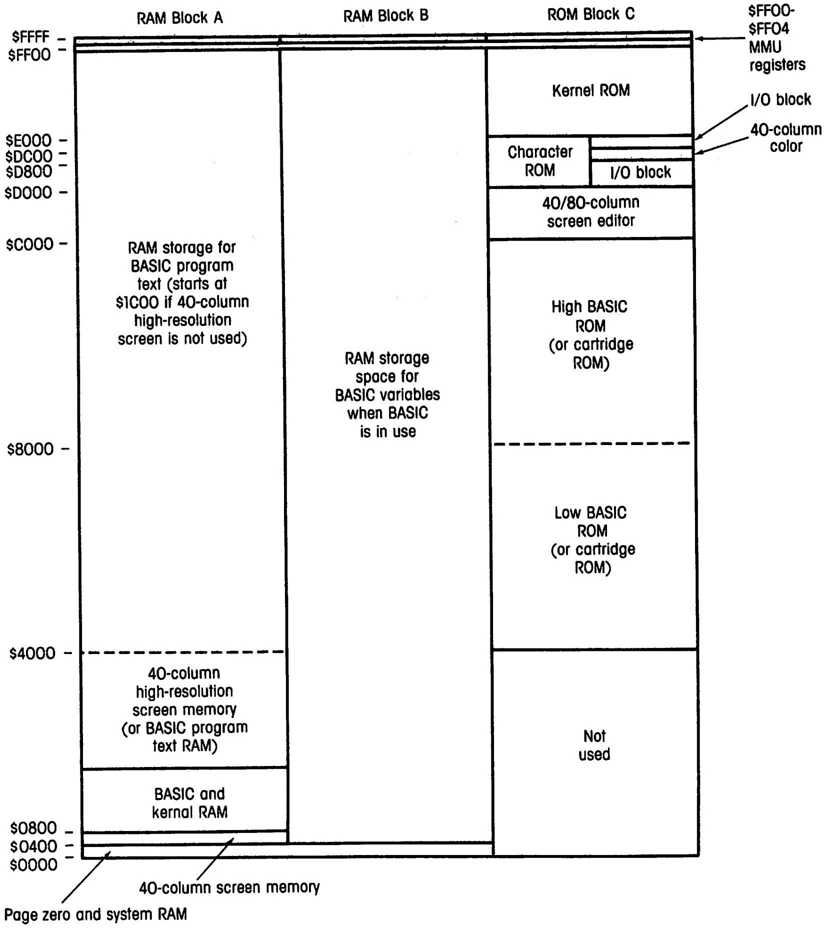

RAM Block A
RAM Block B
ROM Block C
$FF00-$FF04 MMU registers
$FFFF
$FF00
$E000
$DC00
$D800
$D000
$C000
$8000
$4000
$0800
$0400
$0000
Kernel ROM
I/O block
40-column color
Character ROM
I/O block
40/80-column screen editor
RAM storage for BASIC program text (starts at $1C00 if 40-column high-resolution screen is not used)
RAM storage space for BASIC variables when BASIC is in use
High BASIC ROM (or cartridge ROM)
Low BASIC ROM (or cartridge ROM)
40-column high-resolution screen memory (or BASIC program text RAM)
Not used
BASIC and kernal RAM
40-column screen memory
Page zero and system RAM

Figure F-2
Four most commonly used memory banks

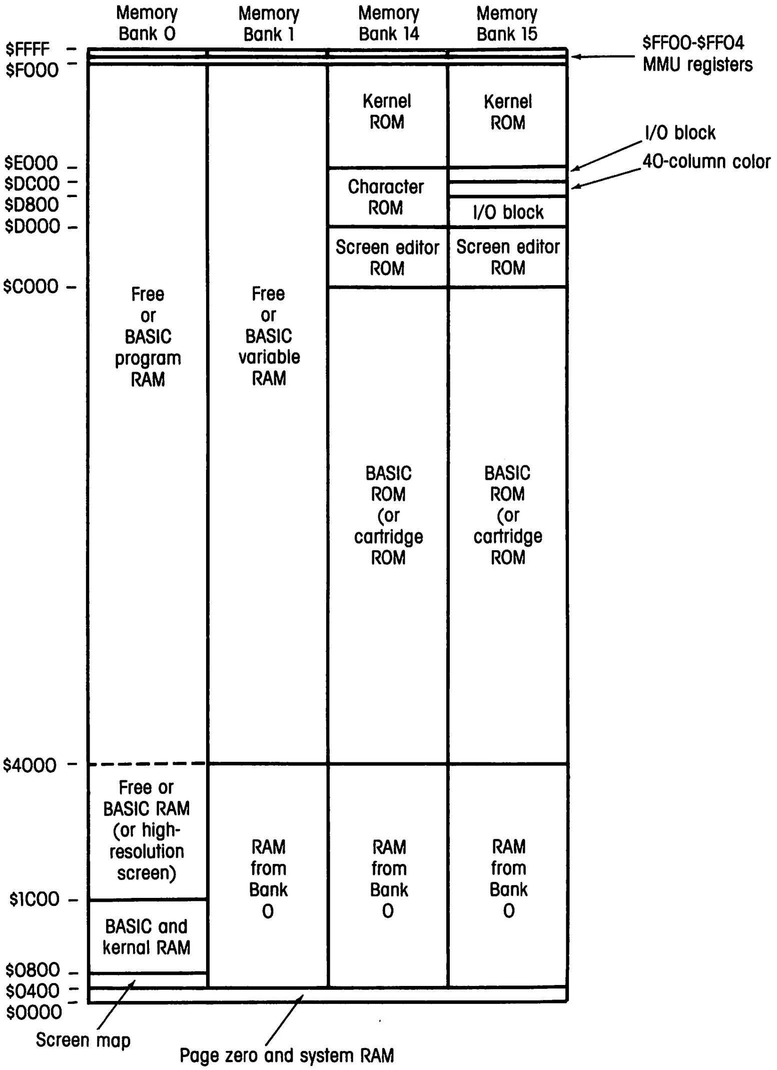


Figure F-3
C-128's CP/M memory map

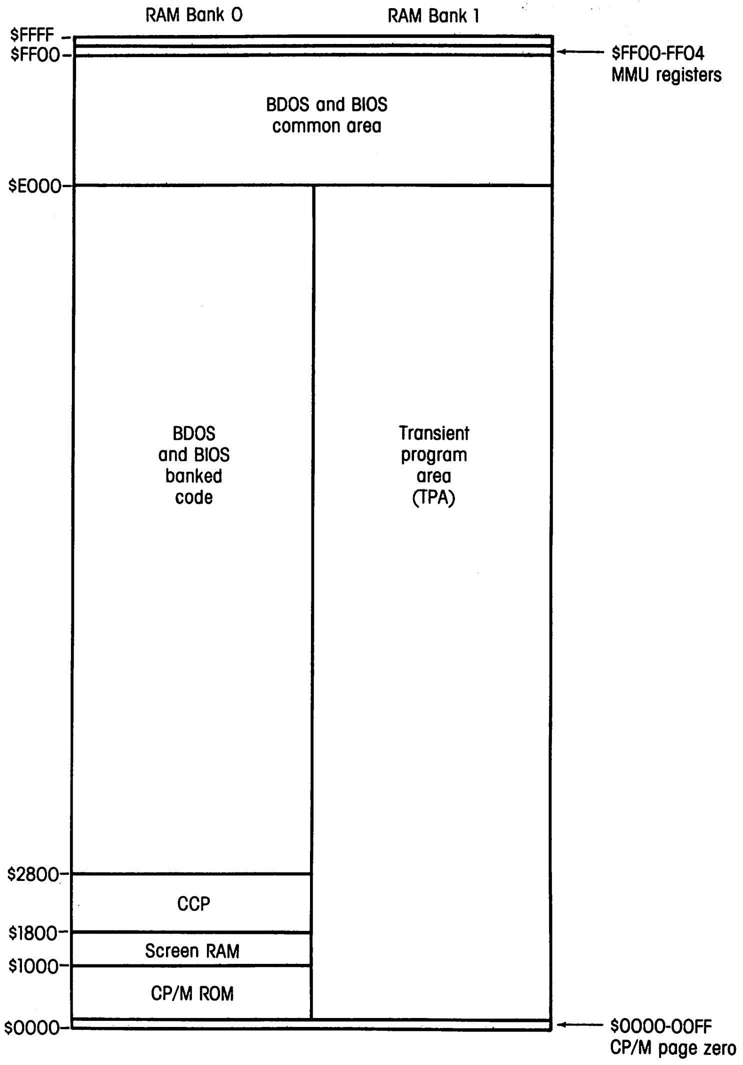


Figure F-4
C-128's C-64 memory map

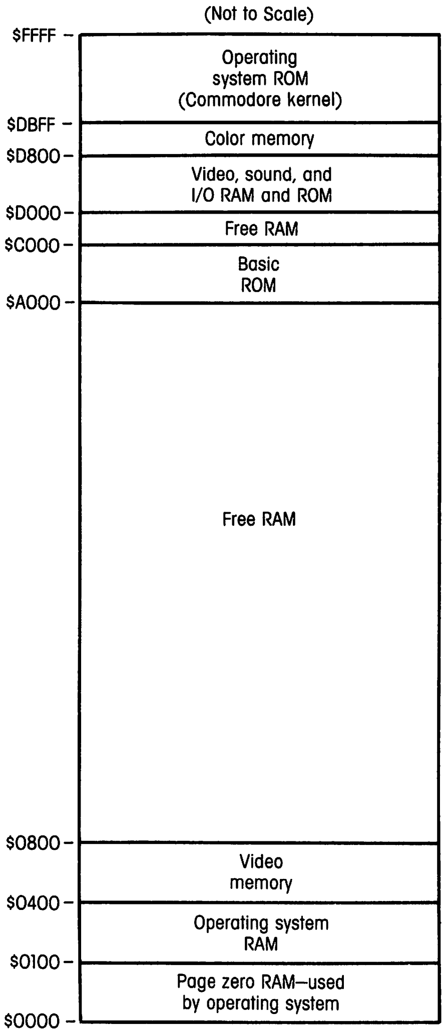
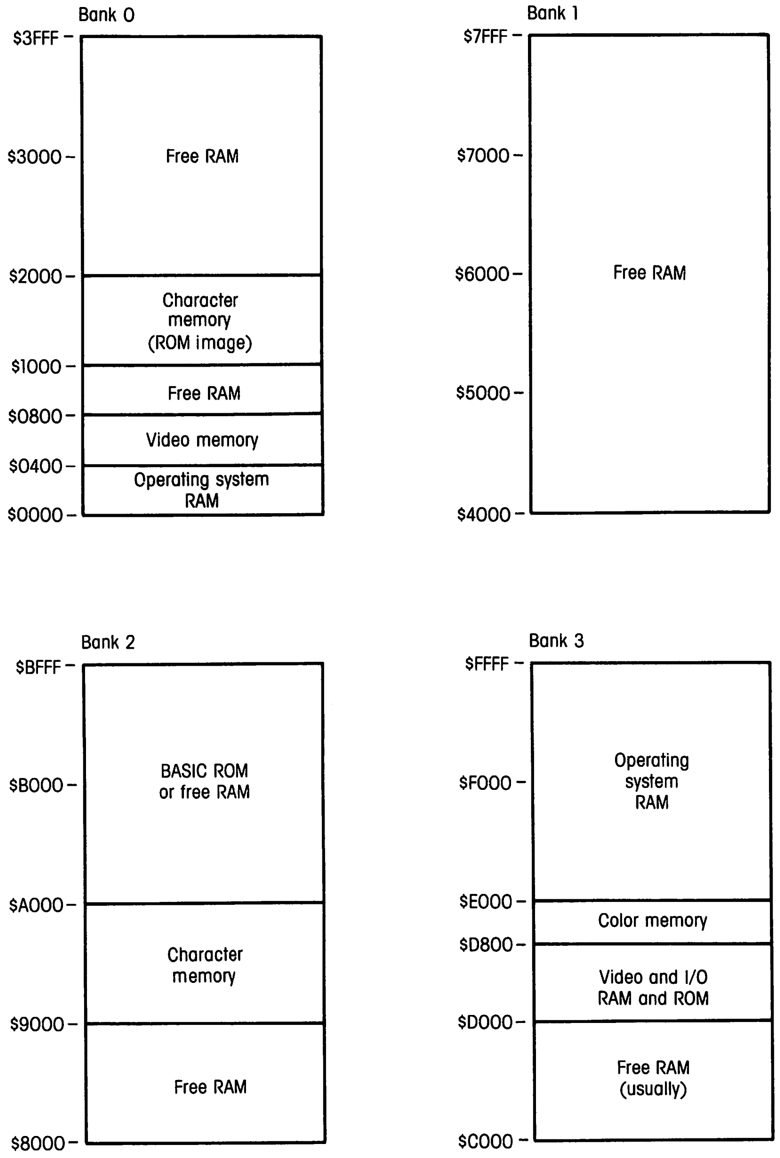


Figure F-5
C-64's four-bank memory map



Bibliography

Abacus Software, Inc. *Commodore CP/M 128 User's Guide.* Grand Rapids, MI: Abacus Software, Inc., 1985.

______. *Commodore 128 Internals.* Grand Rapids, MI: Abacus Software, Inc., 1985.

______. *Commodore 128 Tricks & Tips.* Grand Rapids, MI: Abacus Software, Inc., 1985.

Andrews, Mark. *Apple Roots: Assembly Language Programming.* Berkeley: Osborne McGraw-Hill, 1986.

______. *Atari Roots: A Guide to Atari Assembly Language.* Chatsworth, CA: Datamost, 1984.

______. *Commodore 64/128 Assembly Language Programming.* Indianapolis: Howard W. Sams & Co., Inc., 1985.

______. *The Commodore 64 User's Guide.* New York: Macmillan, 1983.

Bantam Books. *Commodore 128 Programmer's Reference Guide.* New York: Bantam Books, 1986.

Commodore Electronics Limited. *Commodore 128 Personal Computer System Guide.* West Chester, PA: Commodore Electronics Limited, 1985.

COMPUTE! Publications, Inc. *Compute's 128 Programmer's Guide.* Greensboro, NC: COMPUTE! Publications, Inc., 1985.

______. *Mapping the Commodore 64.* Greensboro, NC: COMPUTE! Publications, Inc., 1984.

Findley, Robert. *6502 Software Gourmet Guide & Cookbook.* Rochelle Park, NJ: Hayden Book Co., Inc., 1979.

Heiserman, David L. *Commodore 128 Reference Guide for Programmers.* Indianapolis: Howard W. Sams & Co., Inc., 1986.

Howard W. Sams & Co., Inc. *Commodore 64 Programmer's Reference Guide.* Indianapolis: Howard W. Sams & Co., Inc., 1983.

Leventhal, Lance A. *6502 Assembly Language Programming.* Berkeley: Osborne McGraw-Hill, 1979.

______. *6502 Assembly Language Subroutines.* Berkeley: Osborne McGraw-Hill, 1982.

Maurer, W. Douglas. *Apple Assembly Language.* Rockville, MD: Computer Science Press, Inc., 1984.

Onosko, Tim. *Commodore 64: Getting the Most Out of It.* Bowie, MD: Robert J. Brady Co., 1983.

Wagner, Roger. *Assembly Lines: The Book.* Santee, CA: Roger Wagner Publishing, Inc., 1984.

Zaks, Rodnay. *Programming the 6502.* Berkeley: Sybex, 1983.

Index

MORE FROM SAMS

□ **Experiments in Artificial Intelligence for Microcomputers (2nd Edition)** *John Krutch*
Duplicate such human functions as reasoning, creativity, problem solving, verbal communication, and game planning. Sample programs furnished.
ISBN: 0-672-22501-8, $14.95

□ **Commodore 64® Programmer's Reference Guide** *Commodore Computer*
Includes a complete dictionary of all Commodore BASIC commands, statements, and functions. BASIC program samples then show you how each item works. Mix machine language with BASIC and use hi-res effectively with this easy-to-use guide.
ISBN: 0-672-22056-3, $19.95

□ **Commodore 64® Troubleshooting & Repair Guide** *Robert C. Brenner*
Repair your Commodore 64 yourself, simply and inexpensively. Troubleshooting flowcharts let you diagnose and remedy the probable cause of failure. A chapter on advanced troubleshooting shows the more adventuresome how to perform complex repairs. Some knowledge of electronics is required.
ISBN: 0-672-22363-5, $19.95

□ **Commodore 1541 Troubleshooting & Repair Guide** *Mike Peltier*
This guide presents the theory and general operation of the disk drive and points out some of the common problems that you might encounter.
ISBN: 0-672-22470-4, $19.95

□ **Computer Dictionary (4th Edition)**
Charles J. Sippl
This dictionary of basic computer terms and handbook of computer-related topics includes fiber optics, sensors and vision systems, computer-aided design, engineering, and manufacturing. Clarifies micro, mini, and mainframe terminology. The 1,000 new entries in this edition focus on the RAF classifications: robotics, artificial intelligence, and factory automation. A must for every library.
ISBN: 0-672-22205-1, $24.95

□ **CP/M® Primer (2nd Edition)**
Mitchell Waite and Stephen Murtha, The Waite Group
This tutorial companion to the *CP/M Bible* includes the details of CP/M terminology, operation, capabilities, and internal structure, plus a convenient tear-out reference card with CP/M commands. This revised edition allows you to begin using new or old CP/M versions immediately in any application.
ISBN: 0-672-22170-5, $16.95

□ **Soul of CP/M®: How to Use the Hidden Power of Your CP/M System**
Mitchell Waite and Robert Lafore, The Waite Group
Recommended for those who have read the *CP/M Primer* or who are otherwise familiar with CP/M's outer layer utilities. This companion volume teaches you how to use and modify CP/M's internal features, including how to modify BIOS and use CP/M system calls in your own programs.
ISBN: 0-672-22030-X, $19.95

SAMS

Commodore 128® Assembly Language Programming

Easy to read, written by a Commodore owner for Commodore owners, *Commodore 128 Assembly Language Programming* will teach you how to write commercial-quality programs. If you understand even a little BASIC, this text is for you.

Why assembly language? With machine language (the language taught in most Commodore 128 books) you are limited to writing short routines for BASIC programs. To write high-performance Commodore 128 programs, you need the tool professional program designers use—assembly language.

Commodore 128 Assembly Language Programming:

- Is packed with type-and-run programs and subroutines that you can incorporate into your own programs
- Deals with input/output operations, 40-column and 80-column graphics, and music and sound
- Contains tables and diagrams illustrating important features such as architecture and memory layout
- Covers a host of new features built into the Commodore 128, such as the machine language monitor, bank switching, 80-column text and graphics, and new techniques for interfacing between BASIC and assembly language

With this hands-on guide, you can become an expert Commodore 128 assembly language programmer.

Mark Andrews is the author of the popular Sams book *Commodore 64/128 Assembly Language Programming*. He has written seven other computer books, including three about assembly language. After a stint as consumer electronics columnist with the *New York Daily News*, he became a syndicated electronics columnist. He has taught programming and computer science at the college level, and currently works as a program designer at Pelican Software in Farmington, Connecticut.

0 81262 22541 7

Howard W. Sams & Co.
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

$15.95/22541

ISBN: 0-672-22541-7